Windows Store App Development

C# and XAML

Pete Brown

MANNING

Windows Store App Development

Windows Store App Development

C# AND XAML

PETE BROWN

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2013 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without elemental chlorine.

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Jeff Bleiel
Copyeditor: Linda Recktenwald
Technical proofreader: Thomas McKearney
Proofreader: Elizabeth Martin
Typesetter: Marija Tudor
Cover designer: Marija Tudor

ISBN: 9781617290947
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 18 17 16 15 14 13

brief contents

1 ▪ Hello, Modern Windows 1
2 ▪ The Modern UI 19
3 ▪ The Windows Runtime and .NET 35
4 ▪ XAML 51
5 ▪ Layout 69
6 ▪ Panels 86
7 ▪ Brushes, graphics, styles, and resources 112
8 ▪ Displaying beautiful text 141
9 ▪ Controls, binding, and MVVM 170
10 ▪ View controls, Semantic Zoom, and navigation 211
11 ▪ The app bar 241
12 ▪ The splash screen, app tile, and notifications 265
13 ▪ View states 300
14 ▪ Contracts: playing nicely with others 319
15 ▪ Working with files 342
16 ▪ Asynchronous everywhere 369
17 ▪ Networking with SOAP and RESTful services 388

18 ▪ A chat app using sockets 423
19 ▪ A little UI work: user controls and Blend 465
20 ▪ Networking player location 482
21 ▪ Keyboards, mice, touch, accelerometers, and gamepads 500
22 ▪ App settings and suspend/resume 537
23 ▪ Deploying and selling your app 559

contents

preface *xvii*
acknowledgments *xx*
about this book *xxii*
about the author *xxviii*
about the cover illustration *xxix*

1 ***Hello, Modern Windows*** ***1***
- 1.1 Setting up the development environment 3
- 1.2 Configuring the project 3
  - *The device pane* *5* ▪ *Template solution items* *7*
- 1.3 Create the first Hello World UI 8
- 1.4 Integrating with Twitter 9
  - *The Tweet class* *10* ▪ *Updated UI* *10* ▪ *Code-behind* *11*
- 1.5 Testing on different devices and resolutions 13
  - *Debugging on the Simulator* *13* ▪ *Debugging on a remote device* *14*
- 1.6 Summary 18

2 ***The Modern UI*** ***19***
- 2.1 Design inspiration 20
  - *Direct influences* *21* ▪ *Finding your way* *22*

2.2 Governing principles 23
2.3 Typography 25
2.4 The importance of the layout grid 27
2.5 Design for touch but not only for touch 28
2.6 Modern apps on Windows 8 28
    Consumer and enterprise apps 29 ▪ Key Windows 8 UI elements and states 31
2.7 Device considerations 33
    Desktop or laptop 33 ▪ Tablet and smaller devices 33 Hybrid devices 34
2.8 Summary 34

3 ***The Windows Runtime and .NET 35***
3.1 Windows Store app system architecture 36
    The sandbox 38 ▪ Deployment and the Windows Store 39 The driver model 40
3.2 COM + .NET metadata = WinRT 41
    COM: back to the future 42 ▪ Metadata 44 Projections 46
3.3 Client technologies and languages 47
3.4 A brief tour of WinRT and .NET 4.5 48
3.5 Summary 50

4 ***XAML 51***
4.1 Elements and namespaces 52
    Objects as elements 52 ▪ Namespaces 54
4.2 Properties 56
    Property syntax 56 ▪ Dependency properties 58 Attached properties 61 ▪ Property paths 62
4.3 Object trees and namescope 62
    Object trees 63 ▪ Namescope 66
4.4 Summary 68

5 ***Layout 69***
5.1 Multipass layout—measuring and arranging 70
    The measure pass 71 ▪ The arrange pass 71 The LayoutInformation class 73

5.2 UIElement layout properties 74

Width and Height, plus ActualWidth and ActualHeight 75
Horizontal and vertical alignment 77 ▪ Padding 78
Margins 79

5.3 Layout rounding 80

5.4 Performance considerations 82

Keeping the tree shallow 82 ▪ Caching 83
Virtualization 83 ▪ Sizing and positioning 84

5.5 Summary 84

6 ***Panels 86***

6.1 Canvas 87

Positioning in X,Y space 88 ▪ Controlling the Z position using ZIndex 89 ▪ Sizing child elements 91

6.2 StackPanel and VirtualizingStackPanel 91

Setting the orientation 92 ▪ Sizing children 93
Virtualizing for performance 93

6.3 Grid 94

Defining rows and columns 95 ▪ Adding and positioning elements in rows and columns 97 ▪ Using alignment and margins for sizing and positioning 99

6.4 Creating a custom panel 102

Project setup 102 ▪ The OrbitPanel class 103 ▪ Orbits dependency property 103 ▪ Orbit attached property 105
Custom layout 107

6.5 Summary 111

7 ***Brushes, graphics, styles, and resources 112***

7.1 Brushes 113

Solid-color brushes 113 ▪ Gradient brushes 116
Image brushes 118

7.2 Resources 120

Local and page resources 121 ▪ Application resources 123
Resource dictionaries 123

7.3 Styles 127

Explicit or keyed styles 127 ▪ Style inheritance 128
Implicit styles 130

7.4 Vector graphics 132
Line 132 ▪ Polyline 134 ▪ Paths 135 ▪ Rectangles and ellipses 137
7.5 Bitmap images 137
7.6 Summary 139

8 ***Displaying beautiful text 141***
8.1 Text basics 143
TextBlock 144 ▪ Inlines 146 ▪ Wrapping, ellipsis, and alignment 147 ▪ Character spacing 150 Line spacing 151
8.2 Rich and multicolumn text 153
Rich text 154 ▪ Multicolumn and linked text 157
8.3 OpenType text 160
Ligatures 160 ▪ Stylistic sets 161 ▪ Font capitals 163 Fractions and numbers 164 ▪ Variants, superscript, and subscript 166
8.4 Embedding fonts 167
8.5 Summary 168

9 ***Controls, binding, and MVVM 170***
9.1 The Model-View-ViewModel pattern 172
Using an MVVM toolkit like MVVM Light 174 ▪ The model 175 ▪ The chat data service 176 ▪ The MainViewModel and CameraViewModel classes 178 ▪ The view 180
9.2 Binding primer 183
The source and target 184 ▪ Binding mode 185 Change notification 186 ▪ DataContext 189
9.3 Entering and displaying text 190
Working with the TextBox 191 ▪ Experimenting with the PasswordBox 192 ▪ Spell checking and autocorrect 193
9.4 UI element binding using sliders 194
9.5 Working with lists 197
Observable collections 197 ▪ Items controls 198 Data templates 199
9.6 Making things happen with buttons and commands 200
Button and commands 201 ▪ HyperlinkButton 203 RadioButton and CheckBox 204

9.7 Converting data with value converters 207
9.8 Summary 209

10 *View controls, Semantic Zoom, and navigation* 211
10.1 PhotoBrowser demonstration app setup 213
Creating the project 214 ▪ Creating the Photo model class 215 ▪ Loading pictures using a service class 215 Creating the MainViewModel 217 ▪ Skeleton UI XAML and code-behind 218
10.2 ListView and GridView 220
Vertical lists 220 ▪ Horizontal lists and grids 223
10.3 Grouping with the GridView 226
Grouping in the model and viewmodel 227 ▪ Grouping at the UI layer 228
10.4 FlipView and navigation 231
Viewmodel 232 ▪ Category browser page 232 ▪ Updated MainPage 235
10.5 Semantic Zoom 236
10.6 Summary 239

11 *The app bar* 241
11.1 Project updates 243
11.2 Controls on the bottom app bar 246
Adding and styling buttons 246 ▪ Wiring with commands 250 ▪ Visibility and pinning 258
11.3 Top app bar for navigation 259
11.4 App bar popups and menus 261
11.5 Summary 263

12 *The splash screen, app tile, and notifications* 265
12.1 Splash screens 267
The static splash screen 267 ▪ Extended splash screens 269
12.2 Default tiles on the start page 275
12.3 Secondary or pinned tiles 276
Creating the tile 277 ▪ Activating the app with the secondary tile 281

12.4 Tile notifications or live tiles 284

Simple text notifications 285 ▪ Images in notifications 288
Queuing multiple tile notifications 291

12.5 Toast notifications 294

Creating the notification service 295 ▪ Enabling toast 298

12.6 Summary 299

13 ***View states 300***

13.1 Full, filled, and snapped views 301

13.2 The LayoutAwarePage 303

13.3 The snapped view for the main page 305

13.4 Visual states for view management 307

13.5 Detail pages and app bars 309

Creating an appropriate presentation 310 ▪ Fixing up the app bar 314

13.6 Summary 317

14 ***Contracts: playing nicely with others 319***

14.1 Sharing 320

Sharing your data 321 ▪ Letting others share with you 325

14.2 Letting others search your data 332

Declaring your intentions 332 ▪ The results page and viewmodel 333 ▪ Responding to in-app search requests 338
Responding to external search requests 339

14.3 Summary 340

15 ***Working with files 342***

15.1 Loading files programmatically 343

New demonstration project 343 ▪ File access permissions 347
Storage files and folders 348 ▪ Using a data template selector 351 ▪ Using file queries 354 ▪ Creating files and folders 355

15.2 URI formats 359

15.3 Working with file pickers 361

Using the file open picker 361 ▪ Implementing the file picker source contract 363

15.4 Summary 368

16 *Asynchronous everywhere 369*

16.1 Why asynchronous is important 371

16.2 Working with IAsync* WinRT methods 373

async and await: the simplest approach 374 ▪ Long-form asynchronous operations 376 ▪ Getting progress updates 378 ▪ Canceling the operation 380

16.3 Working with tasks 381

Basic task operations 382 ▪ Canceling the task 384 Converting between WinRT IAsync and Tasks 385*

16.4 Summary 387

17 *Networking with SOAP and RESTful services 388*

17.1 Networking basics 389

Solution setup 390 ▪ Downloading a file with HttpClient 392

17.2 Sharing your model 393

Create the source class library 395 ▪ Create the Modern app–compatible class library 397

17.3 Consuming SOAP services 398

Creating the service 399 ▪ Referencing and using the service 400

17.4 Structuring your client code using MVVM 401

Creating the viewmodel 402 ▪ Creating and wiring up the user interface 404

17.5 Consuming data from RESTful services 406

Creating the RESTful service 407 ▪ Getting data from the service using the viewmodel 410 ▪ Specifying the acceptable data type 411

17.6 Deserializing JSON and XML data 412

XML deserialization using XmlSerializer 412 ▪ JSON deserialization 413

17.7 Updating data using PUT, POST, DELETE, and more 414

17.8 Summary 421

18 *A chat app using sockets 423*

18.1 Chat app viewmodel 425

The MainViewModel class 426 ▪ ChatMessage model class 429

18.2 The user interface 429

XAML skeleton 430 ▪ Styles and resources 431
App bar buttons 432 ▪ Chat app content 433

18.3 Listening for connections 434

18.4 Connecting to the server and sending data 439

Connecting to an endpoint 440 ▪ Sending data 440

18.5 Refactoring for better structure and flexibility 441

The updated ChatMessage class 442 ▪ The IMessageService interface 444 ▪ The TcpStreamMessageService class 447
Updated MainViewModel 454

18.6 Trying out UDP sockets 458

Creating the UdpMessageService class 459 ▪ Listening for connections 460 ▪ Connecting to another machine 461
Receiving and parsing messages 462

18.7 Summary 464

19 *A little UI work: user controls and Blend 465*

19.1 Updated game UI 466

Basic changes 466 ▪ Play field area 468 ▪ Orientation and view states 469

19.2 Designing the ship UI 470

Creating the UserControl 471 ▪ Creating the ship shape in Blend 472 ▪ Adding a label 475

19.3 Building out the ship user control properties 476

Enabling rotation 477 ▪ Setting the color 479
Temporarily testing the Ship control 480

19.4 Summary 481

20 *Networking player location 482*

20.1 Updating the Player model 483

The PlayerLocation class 483 ▪ The updated Player class 484

20.2 The collection of players 485

Initializing the collection 486 ▪ Displaying players with an ItemsControl 488 ▪ Testing the collection 489
Wiring up the collection to service events 490

20.3 Updating the TCP stream message service 491
Updated message service interface 491 ▪ Sending location information 493 ▪ Reading location information 495

20.4 Testing everything 497

20.5 Summary 498

21 *Keyboards, mice, touch, accelerometers, and gamepads 500*

21.1 Making input generic 502
The IInputService interface 502 ▪ A little math help 504 Wiring up the viewmodel 505

21.2 Keyboard input 507
The KeyboardInputService 508 ▪ Virtual keys 510 Adding from the code-behind 512

21.3 Pointer input: mouse, touch, and pen 513
Some more math 513 ▪ A minor modification to the ship user control 517 ▪ The PointerInputService class 517 Adding from the code-behind 519

21.4 Accelerometer input 522
Making sense of the input 523 ▪ Implementing the AccelerometerInputService 524 ▪ Adding from the code-behind 525 ▪ Accelerometer events 526 ▪ Dealing with screen autorotation 526

21.5 Xbox 360 gamepad input and a little C++ 527
Creating the C++ project 527 ▪ Implementing the Controller class 529 ▪ Creating the IInputService wrapper 532 Adding from the code-behind 534 ▪ Compiling and deploying 534

21.6 Summary 535

22 *App settings and suspend/resume 537*

22.1 App settings UI and architecture 538
Creating the settings infrastructure 538 ▪ Creating a settings UI 542

22.2 Persisting and using settings 550
Loading and saving settings values 550 ▪ Acting on the options 553

22.3 Suspend and resume 554

Suspending your app 554 ▪ Resuming activity 555

22.4 Summary 557

23 *Deploying and selling your app 559*

23.1 Testing for certification 560

23.2 Sideloading for testing purposes 563

Packaging an app for sideloading 563 ▪ Getting a developer license without Visual Studio 565 ▪ Installing the sideload app package 566

23.3 Enabling trial mode 567

Creating the mock license data for testing 567 ▪ Checking the license state 569

23.4 Listing your app in the Windows Store 570

Getting a Windows Store account 570 ▪ Reserving an app name 570 ▪ Submitting the app for review and approval 571

23.5 Summary 573

index 575

preface

I've been programming for fun since seventh grade in 1984 and professionally since around 1991/1992. During that time, I've see a lot of change. In the '80s, as the personal computer industry was trying to settle, there were dozens of completely incompatible (both software- and hardware-wise) computers available to the public. In my small group of friends, some owned Commodore 64s, some Commodore VIC-20s, a couple of Apple II variants, a TRS-80 or two, and a couple of others I can't recall. My middle school (properly called a junior high school in Massachusetts) was equipped with some DEC VT-102 Robins, a handful of Commodore VIC-20s (with their disk drive on a serial A/B switch to share between different computers), and a number of Commodore 64 computers. My high school had an Apple IIgs, a couple of Apple IIe computers, and several Apple II computers. Later, they got an Apple IIc and several knockoff Apple clones as well as a lone black-and-white Macintosh. The few computer-literate teachers had access to a handful of IBM PCs to do the serious work of tracking student grades and whatnot. The computer camp I attended in tenth grade used DEC Rainbows and Commodore PETs. The computer competition I attended in tenth grade required knowing Unix and C.

Over the span of four years (seventh grade to tenth grade), I had to learn how to program in multiple incompatible dialects of BASIC and become proficient in multiple different operating systems just to be able to sit down at any given machine and do something useful. (One very interesting trait of these computers, as has been pointed out by others, is that you used to have to choose *not* to program. Programming was the default. More on that some other time.) Later, as a professional just a couple of years out of high school, at a single job I had for a bit under four years, I had to know how to use dBase, FoxPro, Borland Delphi, Borland C++, PowerBuilder, Visual Basic 3, QBasic, QuickBasic, and much more. Oh, and I had to be able to set up the Novel Netware 4 network at the office and convert everyone from dumb terminals to DOS and Windows 3.x PCs. The languages were different, the UI layers were different. There was little to no compatibility between any of these packages.

Change was the norm. It was expected.

Fast-forward to today. As developers, we've never had more pressure on us to be productive, but at the same time, we've never had the longevity of tools, platforms, and languages that we have today. If you started with .NET 1.0 or the alphas/betas (as I did), you've been able to use the same programming language and core runtime for almost 13 years. If you've been a Java programmer, you can claim an even longer run. The only people who had those kinds of runs in the past were FORTRAN and COBOL programmers.

Lately, things have begun to change a bit more. To keep up with the demands of users and the heavy competition in the mobile space, we're seeing programming languages and underlying platforms rev more frequently. A natural consequence of this is deprecation or sunsetting of platforms that don't fit the new interaction models and the emergence of newer API sets, compilers, and more. The JavaScript space has arguably had the most rapid innovation, with new tools and libraries emerging seemingly daily. Many of those have, over the years, completely altered the language in ways that would make modern JavaScript completely foreign to programmers who learned it 10 or even 5 years ago.

On the Windows side, we've seen some amazing work in the .NET and XAML space. Interestingly, despite the changes of underlying platforms and the names of the products, .NET and XAML have remained far more compatible than many other platforms over the years. If you started learning XAML with WPF (or a year later with Silverlight), modern Windows Store XAML will easily become familiar, much like learning BASIC on the Commodore 64 and then learning to program the Apple II. Sure, the PEEK and POKE locations may be different, and there are a few other syntax differences, but there's far more that's compatible than incompatible.

As someone who has made a career of .NET since the first time I gave the two-day .NET 1.0 seminars in the .NET 1.0 alpha days, it's heartening to see that my C# skills are still just as valid today as they were 13 years ago. I'm also happy to see that my investment in XAML starting back in 2006 has served me well across every client platform Microsoft has created. By combining XAML and C#, I can code for the Windows Phone, Windows Store, and the desktop. If I stick to just C#, I can code for everything from tiny ARM microcontrollers on Netduino and Gadgeteer all the way up to massive servers. Through all of this, I'm staying within a tightly focused sphere of development that centers on Visual Studio and C# (or VB, if you prefer).

That's a solid return on investment.

As developers, we tend to focus on the differences in the Microsoft platforms. It's just natural, because it's those differences that give us headaches and make us take up hobbies that involve close encounters with our mortality. But the very fact that we can focus on those differences shows how compatible these platforms are.

For fun, I like to code on microcontrollers. To varying degrees, I've learned ARM with C, AVR with C and C++, PIC with C, NETMF with C#, and a little Arduino. Each of these used completely different IDEs; each uses completely different toolchains. Each time I try to learn another microcontroller, there's very little practical knowledge I can port from one to the other. The registers are all different, the libraries are completely different, and, of course, the IDEs are completely different. This means I've not been able to ramp up on any one platform (with the exception of NETMF because of C#) in a short amount of time; each has been a huge investment in after-the-kids-go-to-sleep time. Few of the IDEs have usable IntelliSense, and help files are almost never in sync with the APIs. It's a lot of trial and error—just getting LEDs to blink on a board feels like a huge accomplishment.

When it came time for me to learn how to write Windows Store apps, I found I had far less to learn than I would have had I been a developer using another platform. Despite WinRT replacing some of the features of .NET, it all felt very familiar. C# worked just as it has all along. Visual Studio was instantly familiar. I can use most of .NET, and the parts that have been replaced by WinRT feel just like .NET.

I'm glad I made the decision, all those years ago, to invest in learning VB3-6 and then C#. I'm also glad I moved from Windows Forms to XAML (WPF and then Silverlight) back in the mid-2000s. Both of these decisions have served me well and will continue to serve me well as Microsoft advances the platforms to better meet the needs of users and to better compete in the marketplace. As a developer, you too should feel confident that although individual products fall out of favor from time to time, your investment in core programming skills, and the higher-level .NET skills beyond that, continues to be just as useful, relevant, and marketable today as it has been over the past decade.

Viva la C#!
Viva la XAML!
Viva el desarrollador!

acknowledgments

My name is on the cover, but technical books like this require a whole team to complete and publish. I'd like to thank the following:

- The various Windows and Developer Division product teams who helped with clarifying just how things work under the covers and who were open to my rather detailed questions.
- Tom McKearney, who has managed to tech review another entire book, and who has provided me with someone to blame if there are any problems with the code listings.
- My friends at Manning Publications: people like Mary Piergies, Linda Recktenwald, Elizabeth Martin, and Jeff Bleiel, who all helped ensure this book is as good as possible and written in one grammatically correct voice.
- The reviewers of the manuscript at various stages of its developent. Your feedback was much appreciated: Brian T. Young, Daniel Martin, Dave Arkell, Dave Campbell, Gordon Mackie (aka Joan Miró), Ian Randall, Krishna Chaitanya Anipindi, Paschal Leloup, Patrick Hastings, Patrick Toohey, Richard Scott-Robinson, Roland Civet, Rupert Wood, and Todd Miranda.

Unique in these thanks is my editor, Jeff Bleiel. This is the third book I've worked on with Jeff. He is my editorial interface with Manning and my continued mentor as an author. Jeff made a positive contribution to this book and to my writing in general.

As with my other books, I'd like to thank my mum for making sure that I knew the difference between "you're" and "your" and that spelling always counts.

Most importantly, I'd like to acknowledge the contribution of my wife, Melissa, and my children, Ben and Abby. Writing a book takes an enormous amount of time, during which I'm not helping around the house, entertaining my children, or otherwise being good company. Thank you to my family for continuing to support me through another book when all my friends are telling me, "Dude, you should be writing apps. You'll make a lot more money."

Finally, thanks to you, my readers. I wrote this and continue to support you in the hopes that I can help you succeed and create awesome apps.

about this book

The goal of this book is to take you, the developer who is at least a little familiar with C# and .NET, and help you become an awesome Windows Store app developer, regardless of which version of Windows you use for building Windows Store apps. If you're already an awesome Windows developer familiar with Windows Store apps, WPF, or Silverlight, I've included deep topics to help you learn more about the platform and how things work under the covers.

After you've read this book, you should be able to confidently design, develop, and deliver Windows Store apps. To facilitate the learning process, I've structured the book to get you developing as soon as possible, while providing quality, in-depth content and several functional apps you can learn from or build on.

Within each chapter, I've included a collection of devices to help you build a firm understanding of the XAML UI platform for Windows. The following list explains how each device helps along the journey:

- *Figures*—Visual depictions that summarize data and help with the connection of complex concepts. Most of these are annotated to call out important details.
- *Code snippets*—Small, concise pieces of code primarily used for showing syntactical formats. You're usually not expected to type these in and compile, because they're incomplete.
- *Code listings*—Code that you can type into your project in Visual Studio. In many cases, it will take multiple code listings to build a working example.
- *Tables*—Easy-to-read summaries.

In addition to these learning devices, my personal site, http://10rem.net, complements the information in this book and often goes into deep detail in specific areas.

Audience

This book is intended for developers who want to create great apps for the Windows Store or for sideloading within an enterprise.

Though XAML provides numerous avenues for interactions with designers, this book primarily targets people who live and breathe inside Visual Studio. With the deeper integration of Blend with Visual Studio, I've included some developer-focused material on working with Blend later in this book.

In addition, and more important, this book assumes you have a background using the .NET Framework and Microsoft Visual Studio. Although we'll be using C# as the primary development language, we won't be reviewing the C# language or explaining basic programming constructs such as classes, methods, and variables.

Experience with Silverlight or WPF will help speed you through the XAML concepts but isn't a prerequisite for this book.

The bits: what you need

This book provides ample opportunity for hands-on learning. But it also provides a great deal of flexibility by allowing you to learn the material without using the hands-on content or optional tools. You'll find it equally valuable to read this book at the computer, on the train, or wherever else you happen to read.

If you want to get the greatest value out of this book and sit down and code or design, here's what you'll need:

- A PC with Windows 8 or higher installed. The examples in this book were originally developed on Windows 8 but will also work with later versions of Windows. You must develop for Windows on Windows. Although it can work, I don't recommend doing this inside a virtual machine because you will run into issues with the Simulator (which is a remote desktop connection to the same machine), and on lower-end machines, performance will suffer.
- You can use the latest recommended version of Visual Studio that will compile for the version of Windows you are using. For Windows 8, here are the recommended versions:
  - Microsoft Visual Studio 2012 Pro or better for both Windows Store and web development, or the free Microsoft Visual Studio 2012 Express for Windows Store apps and, for the networking examples, the free Microsoft Visual Studio 2012 Express for Web.
  - Blend for Visual Studio 2012 or higher. Use the version that's shipped with the version of Visual Studio you're using.

You'll find links to all of these tools, as well as any information on updates, at http://dev.windows.com.

Above and beyond your development PC, you may also find the following optional items useful:

- A Microsoft Surface with Windows Runtime (WinRT) or other ARM-based touch tablet to test compiling and deploying to other architectures.
- Any tablet or other PC with a Windows-recognized accelerometer for testing accelerometer input. If you have a Microsoft Surface, it will serve you well here.
- A wired Xbox controller for Windows for the C++ integration example.
- A device with a Windows-recognized touch screen. This will help, of course, with the touch screen examples. If you have a Microsoft Surface, it will also fill this requirement.
- A second PC for testing the peer-to-peer networking examples. If you have a Microsoft Surface or other tablet running Windows 8+, you can use that.

A note on versions

You'll note a lot of "or the latest version" comments in the software requirements for this book.

The days of waiting three years for an update to Windows and the related development tools have passed. Microsoft recognizes that agility in delivery is as important as individual features. These days, Windows and Visual Studio often update too fast for any publication (or developer, for that matter) to keep up.

For those reasons, the source code for this book will be kept as up to date as much as possible with the latest versions of Windows and tools until the next major revision of this book has been made available. As part of that, if there are any breaking changes in the next version of Windows after Windows 8.1, these will be called out with information made freely available to the purchasers of this book.

With the ongoing Windows commitment to backward compatibility, you can feel safe that the code and techniques you learn for Windows 8 will be applicable to future versions of Windows as well.

Roadmap

WinRT XAML is a brand-new platform but with strong roots in the XAML + .NET platforms that preceded it. I've endeavored to arrange the topics in this book in such a way as to start with a simple example, cover all the basics and theory, and then build out some apps while exploring the more complex topics.

There are no formal sections in this book, but if you squint your eyes a little, you can logically group the chapters as follows:

WINDOWS

It's important to cover the basics up front. Not only are these the technical basics, but they're also the overall themes of the platform and the reasons we made certain decisions. Starting with Windows 8, we have a brand-new set of APIs and design approaches to become familiar with. Chapter 1 introduces Windows and the concepts behind it. Chapter 2 discusses the modern UI, along with its standards and history. Finally, chapter 3 covers the Windows Runtime, what it is, how it has been designed, and why it's so important for Windows Store apps.

XAML AND BASIC CONTROLS

Next, we dive right into XAML. Chapter 4 covers all the basics of XAML, plus a number of topics of interest to advanced developers. Chapter 5 goes into detail on the layout process. This is an important topic for both beginners and advanced developers because so much performance and functionality are affected by the layout engine. Building on that information, chapter 6 covers the commonly used panels, such as the `Grid` and `Canvas`. Chapter 7 deals with graphics, images, and resource management.

One thing you'll learn early on is the importance of text in Windows Store apps. Chapter 8 goes into depth on text, showing how to create beautiful content using text.

Finally, this group wraps up with coverage of binding and controls in chapter 9. I decided to go right into MVVM (Model-View-ViewModel) at this point as well, because it's a good pattern to get used to.

WINDOWS 8 UI SPECIFICS

Although XAML is common and highly compatible across many platforms, Windows Store apps have access to some controls, layouts, and interaction patterns that are unique to the platform. Chapter 10 covers the most important of these controls and how they fit in with navigation. I also cover the Semantic Zoom pattern and control. Chapter 11 covers the modern analog to the toolbar: the app bar. Chapter 12 covers the app's tile and the notification system built into Windows. Chapter 13 covers the important view states all apps must support (snapped view, portrait, landscape, and filled mode).

Throughout all of these chapters, we'll build a PhotoBrowser app that showcases the features.

INTEGRATION WITH THE OS, SERVICES, AND OTHER APPS

Windows Store apps have a set of standardized, user-driven mechanisms for integrating with each other. Collectively, these are called *contracts* and are what are invoked when you use charms such as Search and Share. Chapter 14 specifically covers how to use searching and sharing in the context of the Photo Viewer app.

Chapter 15 then shows how to work with files and file pickers. Files access is the one area that's perhaps the most different from all other XAML and .NET implementations. Throw out everything you've learned about file access in the past, and learn the new (and arguably much better) approach introduced with Windows 8.

NETWORKING AND THE CHAT/GAME PEER-TO-PEER APP

Connected apps are the norm, so several chapters in this book are dedicated to networking. Before jumping into networking, chapter 16 covers the important async patterns used in Windows Store apps. Chapter 17 then starts the coverage of networking by showing how to work with SOAP and RESTful services. This is the type of bread-and-butter stuff any connected app will need. Chapter 18 gets into the more advanced topic of socket communication. As part of the coverage of sockets, we'll start the second app of this book: a chat and peer-to-peer game app involving spaceships.

Chapter 19 helps flesh out the socket app by showing how to create a player control using Blend for Visual Studio. Chapter 20 helps glue the ship and sockets together by showing how to send meaningful information across the socket connection.

Then, because games with just one type of input are a bit boring, we'll implement all the major types of human and device input. In chapter 21, starting with the basics of keyboard input, we'll make the ship move around on the screen and across the network. We'll get into mice and touch-based interaction. Then, we'll make the app accelerometer-aware so you can tilt the device to move the ship around. Finally, we'll implement a C++ WinRT extension library to make use of an Xbox gamepad. Lots of fun stuff here!

WRAPPING UP

To wrap up this book and put a nice bow on what you've learned, I cover the app lifetime, including suspend and resume, and the related app settings in chapter 22. At this point, you'll have a great foundation for building your own apps.

Then, moving out of the source code and into the Store, in chapter 23 I cover how to prepare your app for Windows Store submission and how to get it into the Windows Store.

Code conventions and downloads

You can find the source code for all the examples in the book at www.manning.com/WindowsStoreAppDevelopment.

The following conventions are used throughout the book:

- Italic typeface is used to introduce new terms.
- Courier typeface is used to denote code samples, as well as elements and attributes, method names, classes, interfaces, and other identifiers.
- Code annotations accompany many segments of code.
- Code line continuations use the ➥ symbol.

Author Online

The purchase of *Windows Store App Development* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/WindowsStoreAppDevelopment. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The author online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the author

Pete Brown works for Microsoft as a Technical Evangelist on the Client and Devices team in John Shewchuk's "deep tech" (http://bit.ly/DeepTech) Developer Platform Evangelism group. His role is helping developers create high-quality apps for all clients, using Microsoft tools and technologies.

Prior to joining Microsoft in 2009, Pete was an architect, engagement manager, and user experience designer at a consulting company in the Washington, DC, area, where he focused on Silverlight and WPF development. During that time he was also an INETA speaker, a Microsoft WPF MVP, and a Microsoft Silverlight MVP.

As one of only a few remote workers in corporate Microsoft, Pete has a lot of Lync/Skype webcam meetings and enjoys the stunned look he gets whenever people see his home office in the background. If a nuclear submarine and a radio station had a child near an anime convention staffed by modular synth addicts working on Commodore 64s, it would be only slightly less geeky.

Pete enjoys playing with synthesizers, writing, woodworking, electronics, programming (PC apps as well programs for ARM microcontrollers), making things with no practical use, acquiring huge monitors, cooking processors, and of course, spending time with his wife and two children at their home in Maryland.

about the cover illustration

The figure on the cover of *Windows Store App Development* is captioned "A Traveler." The illustration is taken from a nineteenth-century edition of Sylvain Maréchal's four-volume compendium of regional dress customs published in France. Each illustration is finely drawn and colored by hand. The rich variety of Maréchal's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. On the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Maréchal's pictures.

1

Hello, Modern Windows

This chapter covers

- Building your first Windows 8 app
- Getting a developer license
- Using the Simulator
- Remotely debugging apps

Welcome to the brave new world of Windows Store app development! Getting into a new platform as it first emerges is always exciting. I love learning new things and targeting new platforms. I like being an early adopter. Sure, not everything is quite as fleshed out as you may want when you get in early, and it sometimes feels like the Wild West, but the satisfaction you get from that head start almost always makes it worth it.

Plus, quite frankly, it's much easier to learn something new while it's still small enough to be digestible. Looking back at Silverlight, I learned it at version 1.1a when it was just a baby. Had I picked it up at version 4 or 5 when the API surface area was 10 times as large, it would have taken me forever to learn all its ins and outs. Platforms invariably grow in scope and capability. The earlier you learn them, the less you have to take in all at once.

Because XAML and C# are one of the combinations of technologies you can use to develop for the Windows Store, if you've developed applications for the Windows Phone or desktop using Silverlight, you'll find yourself well positioned to quickly learn Windows app development. The Windows Runtime XAML stack and C#/.NET side of things were both developed by many of the same people who worked on Silverlight and Windows Presentation Foundation (WPF).

Getting into Windows Store app development now will give you that same head start. Using XAML and C# allows you to lean on past experience if you have it and tons of preexisting XAML and C#/.NET content if you don't.

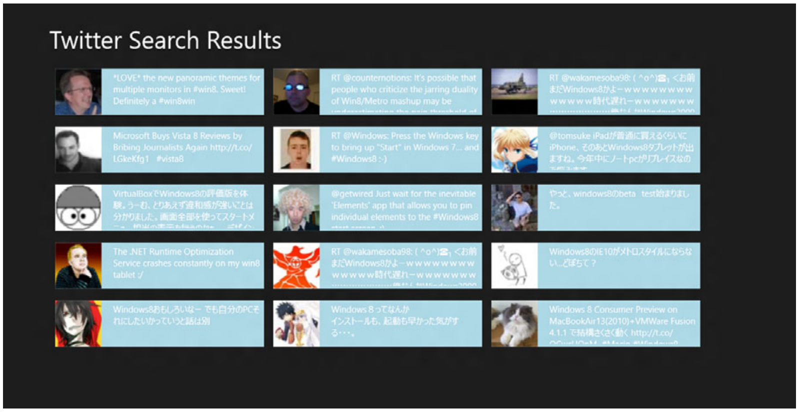
Microsoft is making a serious attempt at capturing the tablet market currently dominated by the iPad and Android devices. If those platforms have taught app developers anything, it's that getting in early with a good app can help spell success for an individual or company.

You picked up this book, so I'm going to assume you're already interested in Windows 8 and don't need encouragement in your selection. Instead, I'd like to take this first chapter and build something right away—just throw you right into the fire. By the end of this chapter, you'll know how to create a simple app and have a working development environment you can use as we dive more deeply into the features in the remainder of this book.

In this chapter, you'll build your very first Windows 8 app using C# and XAML. Neither the application nor the code will be complex. The point here is to acquaint you with a new development and runtime environment and platform rather than produce the mother of all samples.

I'll start by helping you set up your development environment. Once that's complete, you'll create a new project and look at the various configuration settings, property pages, and stock files that are important. Then, I'll show you how to create a very simple Hello World! application. All that application will do is display "Hello World!" on the screen.

From there, you'll modify the application to pull data from Twitter and display it onscreen. When run, the completed application will look like figure 1.1. Despite not



Twitter Search Results

LOVE the new panoramic themes for multiple monitors in #win8. Sweet! Definitely a #win8win

RT @counternotions: It's possible that people who criticize the jarring duality of Win8/Metro mashup may be

RT @wakamesoba98: (^o^)☎┐ <お前まだWindows8かよーｗｗｗｗｗｗｗｗｗｗｗｗｗ時代遅れーｗｗｗｗｗｗｗ

Microsoft Buys Vista 8 Reviews by Bribing Journalists Again http://t.co/LGkeKfg1 #vista8

RT @Windows: Press the Windows key to bring up "Start" in Windows 7... and #Windows8 :-)

@tomsuke iPadが普通に買えるくらいにiPhone、そのあとWindows8タブレットが出ますね。今年中にノートpcがリプレイスなの

VirtualBoxでWindows8の評価版を体験。うーむ、とりあえず違和感が強いことは分かりました。画面全部を使ってスタートメ

@getwired Just wait for the inevitable 'Elements' app that allows you to pin individual elements to the #Windows8

やっと、windows8のbeta test始まりました。

The .NET Runtime Optimization Service crashes constantly on my win8 tablet :/

RT @wakamesoba98: (^o^)☎┐ <お前まだWindows8かよーｗｗｗｗｗｗｗｗｗｗｗｗｗ時代遅れーｗｗｗｗｗｗｗ

Windows8のIE10がメトロスタイルにならない…どぼちて？

Windows8おもしろいなー でも自分のPCそれにしたいかっていうと話は別

Windows 8ってなんか
インストールも、起動も早かった気がする・・・。

Windows 8 Consumer Preview on MacBookAir13(2010)+VMWare Fusion 4.1.1 で結構さくさく動く http://t.co/

Figure 1.1 The completed Hello World! application.

being very pretty, this app follows the new design aesthetic, runs full screen, and supports touch. It uses the new `WrapGrid` layout control and, to many, familiar XAML concepts. If you don't know XAML yourself, don't worry; you will by the end of this book.

1.1 Setting up the development environment

Windows 8 Modern apps can only be developed on Windows 8, so the first thing you'll need to do is install Windows 8. Hardware selection is too large a topic for this book, but you'll want to make sure the screen width is at least 1366 x 768 or larger so you can use snapped views for apps. The machine will need to be based on an x86-compatible processor because Windows RT (ARM) devices can't run Visual Studio.

The ideal situation is to install on the metal or set up a bootable VHD on a typical developer-class machine. Running in a virtual machine is okay, but you'll find the experience frustrating at times, especially when it comes to swiping from the edges or activating elements using the hot corners.

> **YOUR MICROSOFT ACCOUNT** When installing Windows, unless you're using a domain account, make sure you set up a Microsoft account rather than a local account. Microsoft accounts have much better integration with online services, and you'll need one to obtain a developer key. *Microsoft account* is a new name for Windows Live ID (or Passport, if you can remember back that far)—something almost every .NET developer already has.

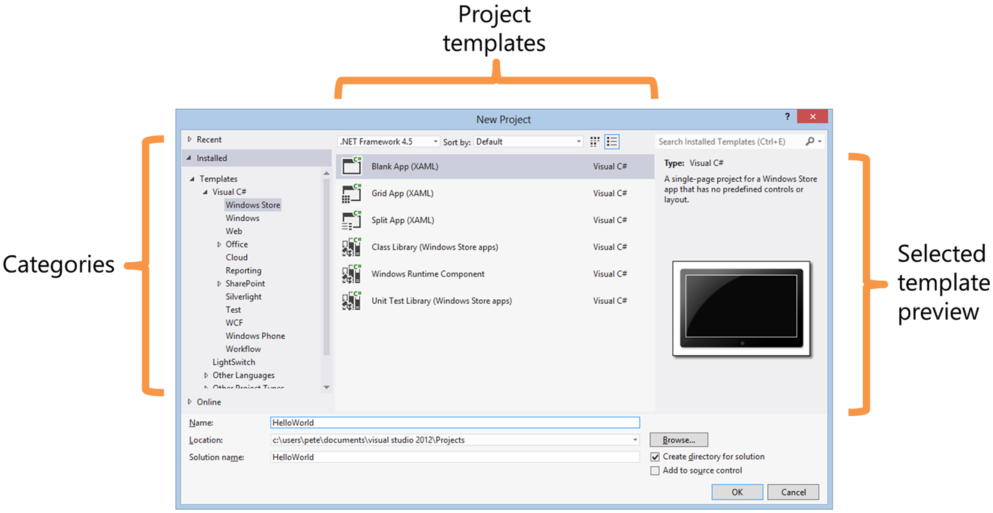
To develop Windows apps, you'll need to have a version of Visual Studio 2012. You can use the free Visual Studio Express for Windows 8 apps, or you can install a higher level and more feature-rich version such as Professional. You can find links to all the important bits at http://dev.windows.com. All the required SDKs and templates are installed automatically with Visual Studio 2012; even .NET 4.5 and the Windows Runtime are already installed on the Windows 8 machine. No additional downloads are required.

You don't need a Windows Store account just yet, but you'll eventually want one of those as well (more on the Windows Store in chapter 23). You can obtain a free developer key automatically the first time you create a Modern Style app in Visual Studio, as you'll see in a moment.

1.2 Configuring the project

In keeping with the spirit of Hello World, your first Windows app will be a simple one designed to let you see the construction and build process from front to back. Don't worry about understanding all the parts of the project just yet; that'll come in the subsequent chapters.

In Visual Studio, select the Start page link (or File menu option) to create a new project. This will be a Visual C# project, a Windows Store app, using the Blank App (XAML) template. The .NET framework version used is 4.5—the only version currently supported for Windows app development. I named it *HelloWorld*, as shown in figure 1.2.


Project templates
Categories
Selected template preview
New Project
Recent
Installed
Templates
Visual C#
Windows Store
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Windows Phone
Workflow
LightSwitch
Other Languages
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank App (XAML) Visual C#
Grid App (XAML) Visual C#
Split App (XAML) Visual C#
Class Library (Windows Store apps) Visual C#
Windows Runtime Component Visual C#
Unit Test Library (Windows Store apps) Visual C#
Browse...
OK
Cancel

Type: Visual C#

A single-page project for a Windows Store app that has no predefined controls or layout.

Name: HelloWorld

Location: c:\users\pete\documents\visual studio 2012\Projects

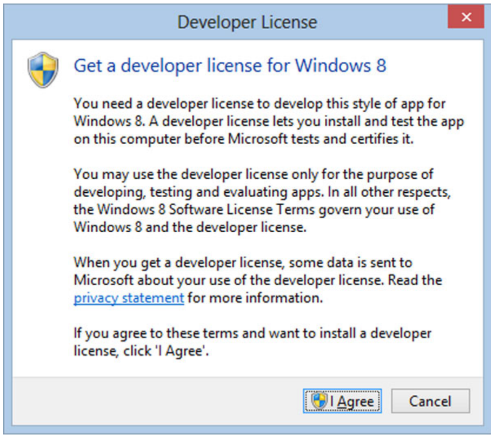
Solution name: HelloWorld

☑ Create directory for solution

☐ Add to source control

Figure 1.2 Creating the Windows XAML app using the Blank App template

Notice the other project templates: Grid App (XAML) and Split App (XAML) in particular. Those are feature-rich templates, much like the business application templates and navigation templates in Silverlight. I'll skip these templates for now, because we'll dive into them later in this book, and there's a lot more to them than we want to get into in chapter 1.


Developer License
I Agree
Cancel

Get a developer license for Windows 8

You need a developer license to develop this style of app for Windows 8. A developer license lets you install and test the app on this computer before Microsoft tests and certifies it.

You may use the developer license only for the purpose of developing, testing and evaluating apps. In all other respects, the Windows 8 Software License Terms govern your use of Windows 8 and the developer license.

When you get a developer license, some data is sent to Microsoft about your use of the developer license. Read the privacy statement for more information.

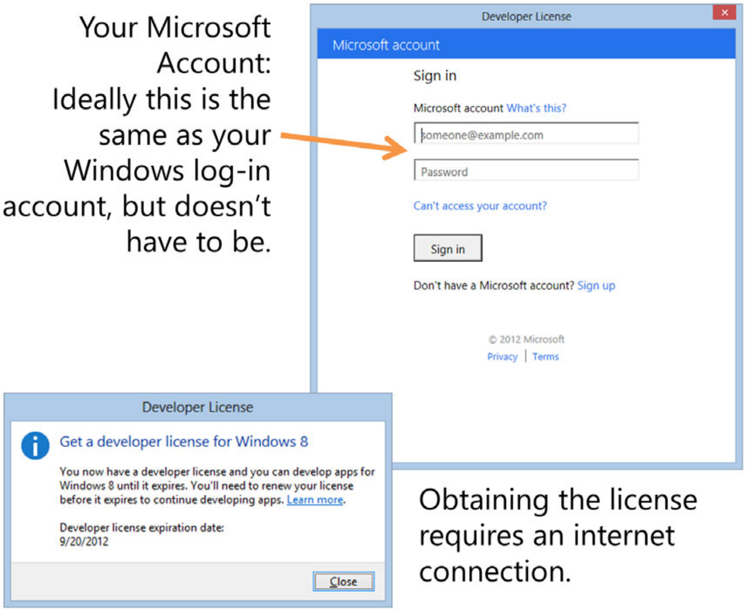
If you agree to these terms and want to install a developer license, click 'I Agree'.

Figure 1.3 Prompt to obtain a developer license

Click OK to create the project. If this is the first time you've created a Windows 8 app on your machine, you'll be prompted to create a developer license, as shown in figures 1.3 and 1.4.

When you first run Visual Studio and try to build a Windows 8 app, you'll have to register for a developer license. This is free and is granted for 30 days for non-store use and 90 days if you have a Windows Store account (these durations are subject to change). Renewing is painless as long as you have an internet connection. It may be a bit annoying to renew every 30 days, but it's a necessary step to make sure that people creating malware or otherwise trying to cause havoc can have their keys turned off.

You can renew your developer license through the command prompt or through Visual Studio at any time should you know you're going to be disconnected from the



Developer License

Microsoft account

Sign in

Microsoft account What's this?

someone@example.com

Password

Can't access your account?

Don't have a Microsoft account? Sign up

© 2012 Microsoft

Privacy | Terms

Developer License

Get a developer license for Windows 8

You now have a developer license and you can develop apps for Windows 8 until it expires. You'll need to renew your license before it expires to continue developing apps. Learn more.

Developer license expiration date:
9/20/2012

Your Microsoft Account: Ideally this is the same as your Windows log-in account, but doesn't have to be.

Obtaining the license requires an internet connection.

Figure 1.4 You'll need to be online when you go to obtain the developer license. If not, you'll be unable to continue creating the Windows 8 app. Don't wait until you're off the grid (in a taxi or one of those planes without Wi-Fi) before obtaining or renewing your license.

internet for a bit. You can find the instructions at http://bit.ly/Win8DevLicense. You may consider automating the renewal to happen every 29 days or some other time period by scheduling a job to do it for you.

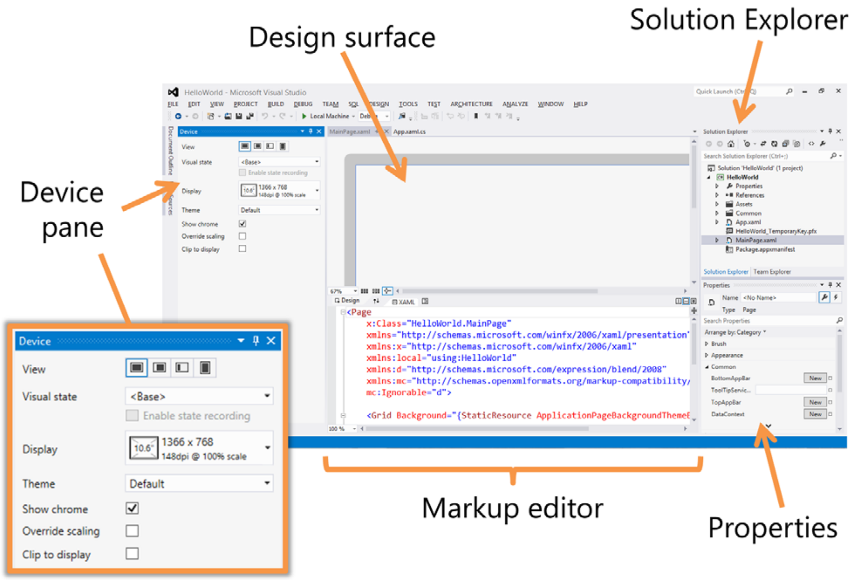
> **TIP** I speak at a lot of events. I assume many of you also give demos to potential clients, managers, and others. If it has been a few weeks since you renewed your developer license, do so before you give that important demo. Nothing will tank your presentation faster than being prompted to renew your developer license when there's no available internet connection. This happened to me on one of the few plane trips I took that didn't have working Wi-Fi. I read a book instead.

If you want to keep things simple, use a single ID for your machine login and your developer license. This isn't always possible, especially in the case of domain-joined machines, so consider it a recommendation and not a hard rule.

Once you have the developer license, you'll be tossed into the IDE with your project loaded.

1.2.1 The device pane

Once the application template is loaded, you'll be presented with the solution. Open MainPage.xaml and you'll see the design surface and the usual developer window



```
<Page
    x:Class="HelloWorld.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:HelloWorld"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeB
```

Visual state <Base>
☐ Enable state recording
Display: 10.6" 1366 x 768 148dpi @ 100% scale
Theme: Default
Show chrome: ☑
Override scaling: ☐
Clip to display: ☐

Figure 1.5 Visual Studio 2012 with the blank project template loaded in the Hello World project. The Device window on the left is new to Windows app development and is a real convenience for screen layout.

panes. One pane, the Device window panel, shown in figure 1.5, stands out as new, however.

> **TIP** Your design surface will likely be black. If not, compile the solution so the design surface can update.

The Device pane (windows are generally referred to as *panes* when docked) to the left of the design surface has a number of buttons to let you try out different views for your application: Full, Filled, Snapped, and Portrait. Under the View buttons, there's the Display selector. This lets you select from common display resolutions. Finally, there are options that control other aspects of the simulated tablet (or touch screen) on the design surface.

> **TIP** If the Device window/pane isn't visible, go to the DESIGN menu and select Device Window. It's not on the VIEW menu, unlike most other dockable windows. Yes, the menus are in all caps. Embrace the case.

All of these options are external to your application code and markup; they don't change anything. They simply let you test your UI layout under a number of different configurations without actually deploying it to a machine of that resolution. Nice!

Feel free to play with the options a bit, but then return them to their defaults when finished. I'm using a test resolution of 1366 x 768 (at 148 dpi), the minimum full-featured Windows 8 resolution, and have the view configured to the full view.

1.2.2 Template solution items

To the right of the design surface, you'll see the Solution Explorer. As a .NET developer, you're almost certainly familiar with this by now. Notice, however, that there are a few more files and folders there than you may be used to.

If you're a Silverlight or WPF developer, App.xaml should be familiar to you. For those new to the platform, this is where the startup code exists and where you can keep styles, templates, and other resources that are shared throughout the entire application.

The Common folder contains the very important StandardStyles.xaml file. You'll use this file, or more correctly, the resources in it, when creating the UI for the applications throughout this book. This file is where the Windows 8 Modern app styles reside.

Don't mess with the standard styles...at first

I mentioned that the StandardStyles.xaml file is important. The note at the top of the file says it all:

"This file contains XAML styles that simplify application development."

These are not merely convenient but are required by most Visual Studio project and item templates. Removing, renaming, or otherwise modifying the content of these files may result in a project that doesn't build or that won't build once additional pages are added. If variations on these styles are desired, it is recommended that you copy the content under a new name and modify your private copy.

So treat this file like you would autogenerated files, and leave it alone until you have more experience in app development. Experienced developers will want to pare down this file before deployment, because the additional styles do add a small amount of load time.

The Assets folder contains the application images. In it, you'll find the app's main and small logo as used in the tiles, the splash screen logo, and the logo to be used in the Windows Store.

Right-click the Hello World project (not the solution) and choose Properties. Most of the options here are familiar, but click the Debug tab on the left. In there you'll see a number of start options. By default, you'll run on the local machine. But there's also an option to start the application on a remote machine. This is key if you want to actively test and debug on a sub-developer-class machine or a machine that simply can't run Visual Studio, like a Windows on ARM tablet. Additionally, there's the option to start the app in the Simulator, which we'll explore shortly.

Next, double-click the Package.appxmanifest file. This is where you'll specify the runtime configuration for your application, including the logos, supported orientations, background color, capabilities (such as needing access to the pictures library), and declarations (such as registering as a picture provider or as a file save picker).

This is also where you can specify your start page's tile background and foreground colors and logos. In short, this is the place where many of the important packaging and deployment details are kept.

1.3 Create the first Hello World UI

The next step is to create a very simple Hello World UI. Open up MainPage.xaml if it isn't already open. Before you do any real work, turn on Show Grid (this is the 20-pixel grid we'll cover in chapter 2) and turn on Snap to Grid. Both controls are at the bottom of the design window, as shown in figure 1.6.

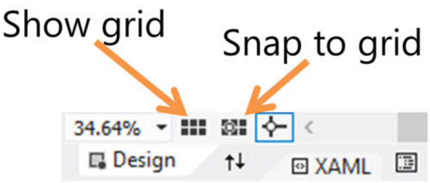


Figure 1.6 The Show Grid and Snap to Grid controls (the six-box grids) on the XAML design surface. If you plan to drag controls onto the design surface, turn on both Show Grid and Snap to Grid by clicking the buttons shown here. These buttons can be found at the bottom of the design surface or between the design surface and the XAML view if you're using the default IDE layout.

Next, you can either drag a `TextBlock` from the Toolbox (pane on the left—you may need to click it or hover over it if it's not already open) onto the design surface and visually align its left and text baseline with one of the major grid lines, or you can simply paste inside the `Grid` element the following XAML:

```
<TextBlock Text="Hello World!"
           Height="42" Width="270"
           HorizontalAlignment="Left" VerticalAlignment="Top"
           FontSize="42" Margin="80,40,0,0" />
```

If you run the application at this point, you should see a nice "Hello World!" message. If you're running the default themes, it will be white text on a black background. You may be wondering how to get out of the application and back to Visual Studio. You have a few options:

- On a single-display system, use Alt-Tab to get back to Visual Studio and keep the app running (you'll use this often during debugging).
- Press Alt-F4 to close the app and then hit the Windows key to get back to Visual Studio.
- Move the mouse to the top of the app until it turns into a hand, and then drag down to close it. This gesture also works with touch, of course.

By design, Windows 8 apps don't have dedicated close buttons; the user uses one of the aforementioned approaches to get out of the app.

Where did the background color come from?

You may have noticed that the root `Grid` has its background color bound to a static resource named `ApplicationPageBackgroundThemeBrush`. If you go hunting around, you won't find that brush defined anywhere. Where is it, then?

This is one of the standard SDK resources, loaded as part of the platform. You don't have to use it, but if you want to have a well-behaved Modern Style Windows 8 app, you probably will want to stick with the default resources. Why? They respond to themes picked by the user, including high-contrast and standard themes.

If you create your own controls, you'll definitely want to support these resources in your default template.

Of course, if you know what you're doing, by all means, just ignore it and don't use it or the other stock resources. There are a number of them, but they're highly optimized so you don't take a performance hit from unused resources at runtime. If you want to see them, a copy of the resources can be found in \Program Files (x86)\Windows Kits\8.0\Include\winrt\xaml\design.

If all you want is a simple Hello World, you can stop here. But if you want to kick it up just a tiny notch and add something meaningful to your Hello World app, read on. It'll still be simple, but rather than just the "Hello World!" text, we'll add a little data from our friends at Twitter. I just can't help myself.

1.4 Integrating with Twitter

Networking is almost always part of a modern application. Windows 8 apps are usually online and always connected (or often connected). Networking in Windows 8 is, therefore, suitably feature rich and central to the platform.

This code is going to call out to Twitter to get a list of Tweets that contain the hashtag[1] "#win8" or "#windows8." It will then deserialize the results into a set of `Tweet` objects, which will then be bound to a `ListView` in the UI. If you've read any of my other books, you'll know this is a consistent example I follow. It'll also be interesting for you to see the differences in how the networking code here compares with the code in my Silverlight books. If you haven't read my Silverlight books, no worries; everything here will be explained.

In this section, you'll start by creating the `Tweet` class to hold the key information about the Tweet. Next, you'll update the UI using a `ListView`, a `WrapGrid`, and an item template to display the Tweets. This is a common approach for displaying data in a XAML app. Finally, you'll throw some code in the code-behind to make the networking calls and parse the data that's returned.

---

[1] I wonder if the child named Hashtag will have a hard time doing an ego search on the internet? Will they sign things #lastname? Yes, someone named their child "Hashtag."

1.4.1 The Tweet class

First, you'll need to create a class to hold the data that will be returned from your Twitter search. Create a folder named Model, and in it create a new class named `Tweet`. The first listing shows the `Tweet` class code.

Listing 1.1 Code for the `Tweet` class in the Model folder

```
using System;

namespace HelloWorld.Model
{
  class Tweet
  {
    public string Message { get; set; }
    public string Image { get; set; }
  }
}
```

A Twitter search returns much more information for each Tweet, but the `Message` and `Image` are all you'll use in this example. They are auto-properties (properties with auto-generated `get` and `set` accessors) in order to make binding possible. You'll learn more about binding in chapter 9.

1.4.2 Updated UI

The next step is to update the UI to provide a nice way to list the Tweets. You'll use a `WrapGrid` as shown in the following listing. Silverlight and WPF developers may remember this as the `WrapPanel`. They're conceptually the same, but the `WrapGrid` only works inside items controls.

Listing 1.2 Updated UI with the `ListBox` of Tweets

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <TextBlock Text="Twitter Search Results"                          Heading
             HorizontalAlignment="Left" VerticalAlignment="Top"
             FontSize="42" Margin="80,40,0,0" />

  <ListView x:Name="TweetList" Margin="80, 100, 80, 80">         ListView control
    <ListView.ItemsPanel>
      <ItemsPanelTemplate>
        <WrapGrid />                                     WrapGrid for
      </ItemsPanelTemplate>                              items layout
    </ListView.ItemsPanel>
    <ListView.ItemTemplate>
      <DataTemplate>
        <Grid Width="360" Height="80" Background="LightBlue">
          <Image Margin="0" Width="80" Height="80"                    Image
                 HorizontalAlignment="Left"
                 Source="{Binding Image}"
                 Stretch="Uniform"/>
          <TextBlock Text="{Binding Message}"                 Message
                     HorizontalAlignment="Left"
                     VerticalAlignment="Top"
                     FontSize="15"
                     Foreground="White"
                     TextWrapping="Wrap"
                     Margin="100,5,5,5"/>
         </Grid>
       </DataTemplate>
     </ListView.ItemTemplate>
   </ListView>
</Grid>
```

In MainPage.xaml, replace everything inside and including the main `Grid` with what you see in this listing, but leave the `Page` declarations and namespaces alone. This listing sets up the UI so that it lays out all elements on a wrapped grid. Each element is a simple rectangular grid with a blue background, a photo to the left, and text on the right.

The final step is to wire everything in the code-behind.

WHY A LISTVIEW INSTEAD OF A LISTBOX? WPF and Silverlight developers know that you can do just about anything with a `ListBox`. In Windows 8 apps, due to plumbing changes for animation and virtualization, the replacement for the `ListBox` is the `ListView`. You can still use the `ListBox` for some scenarios, but you'll find it no longer works with all the layout grids. When in doubt, reach for the new `ListView` control instead.

1.4.3 Code-behind

Later in this book you'll learn best practices for separating code using `ViewModel`s and varying degrees of the MVVM pattern. For this first example, all the code will remain in the code-behind as shown in the following listing. This code goes in the code-behind of the main page (MainPage.xaml.cs).

Listing 1.3 Code-behind for the application's main page

```
using System;
using System.Collections.ObjectModel;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Xml.Linq;
using HelloWorld.Model;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace HelloWorld
{
  public sealed partial class MainPage : Page
  {
    public MainPage()
    {
      this.InitializeComponent();
    }
    private ObservableCollection<Tweet> _tweets =
            new ObservableCollection<Tweet>() ;

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
      TweetList.ItemsSource = _tweets;
      LoadTweets();
    }

    private async void LoadTweets()
    {
      Uri uri = new
         Uri("http://search.twitter.com/search.atom?q=win8+OR+windows8");

      var client = new HttpClient();
      var response = await client.GetAsync(uri);

      string content = await response.Content.ReadAsStringAsync();

      Debug.WriteLine(content);

      var doc = XDocument.Parse(content);
      XNamespace ns = "http://www.w3.org/2005/Atom";

      var items = from item in doc.Descendants(ns + "entry")
                  select new Tweet()
                  {
                    Message = item.Element(ns + "title").Value,
                    Image =
                      (from XElement xe in item.Descendants(ns + "link")
                       where xe.Attribute("type").Value == "image/png"
                       select xe.Attribute("href").Value).First()
                  };

      _tweets.Clear();

      foreach (Tweet t in items)
        _tweets.Add(t);
    }
  }
}
```

Create collection of Tweets

Bind ListView

Make async calls

Display raw data

Parse data

Load into collection

If you haven't used .NET 4.5 yet (or Silverlight 5 with Visual Studio 2012 and the async targeting pack), the `async` and `await` keywords will be new to you. These two keywords help remove a lot of the asynchronous handler code you had to write in the past. Because the Windows Runtime relies so heavily on asynchronous methods, this is a huge boon to Windows 8 developers. What used to take a convoluted set of callbacks can now be reduced to a single `await` statement. Nice!

Other than that, everything there should look familiar to Silverlight developers. The XML parsing works just as it would in any other .NET project. The navigation page method `OnNavigatedTo` looks like those in the navigation templates in Silverlight. The good-old `ObservableCollection` is there and more. Your knowledge and skills port well.

But don't worry if you have no Silverlight experience. Although I'll bring up differences from time to time to further inform readers coming from similar languages and platforms, I don't assume any XAML experience in this book.

Run the application, and you should see a list of Tweets complete with text and images similar to those shown in figure 1.1 at the start of this chapter.

1.5 Testing on different devices and resolutions

So far, we've run everything directly on the development PC. As you know, developer machines aren't representative of the typical end-user machine. In a world of touch devices, hybrid laptops, and tablets, this is even truer.

Because the devices can vary so much in capability, and because installing Visual Studio on every device is both impractical and impossible, there has to be a good way to deploy and debug remotely. One of my favorite things about .NET Micro Framework development is the ability to deploy code to an external device and step through breakpoints on my development PC. Windows Phone developers and developers who remotely debugged apps on the Mac will also be familiar with this Visual Studio capability.

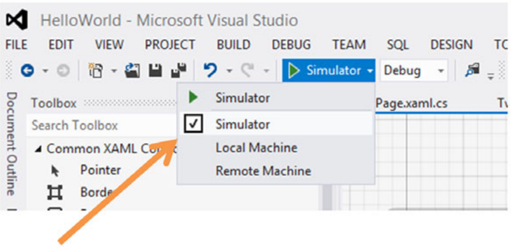
Sometimes all you need to do is try out different resolutions or simulate touch on your non-touch Windows development machine. For those scenarios, the built-in Simulator is the way to go. The Simulator makes it simple to deploy and debug inside a simulated machine, where you control resolution and other parameters.

In this section we'll look at a few different ways to debug the app. First, you'll run it on the Simulator and then move to debugging on a remote machine. If you don't have a second Windows 8 machine handy, that's okay.

1.5.1 Debugging on the Simulator

Visual Studio 2012 comes with a Windows 8 Simulator. The Simulator is actually a remote desktop into the same machine Visual Studio is running on, but it abstracts this and provides another layer of UI interaction. This enables you to simulate touch, gestures, rotation, different screen resolutions, orientations, and more.

Continuing from where we left off, exit the app and change the Debug target to be the Simulator rather than the local machine. You can do this from the project properties or, more easily, through the toolbar, as figure 1.7 shows.



Debug in Simulator

Figure 1.7 The menu option to debug from the Simulator rather than from the local machine. This is great for testing out different resolutions or simulating touch on a non-touch device.

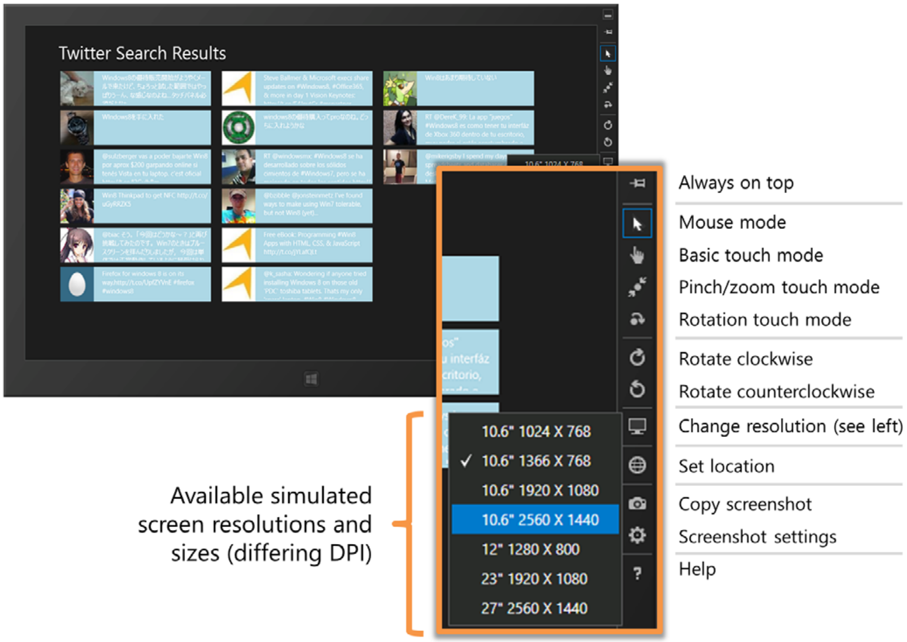

Twitter Search Results
Always on top
Mouse mode
Basic touch mode
Pinch/zoom touch mode
Rotation touch mode
Rotate clockwise
Rotate counterclockwise
Change resolution (see left)
Set location
Copy screenshot
Screenshot settings
Help
10.6" 1024 X 768
✓ 10.6" 1366 X 768
10.6" 1920 X 1080
10.6" 2560 X 1440
12" 1280 X 800
23" 1920 X 1080
27" 2560 X 1440
Available simulated screen resolutions and sizes (differing DPI)

Figure 1.8 The Twitter app running in the device Simulator. The controls on the right provide a number of options for changing the shape of the device and for interacting with it.

Now when you debug the app, it will open up in the Simulator. Figure 1.8 shows the app running in the Simulator and explains the different Simulator controls.

The Simulator is excellent when it comes to testing different resolutions or simulating touch on a non-touch device. That will get you very far for the majority of the apps out there. For the more performance- or feature-dependent apps, you'll want to debug on the actual target device.

1.5.2 Debugging on a remote device

Windows 8 supports both the x86/64 and ARM architectures. Visual Studio will run only on x86/64. Because of that, and because even many x86 tablets simply aren't appropriate for running Visual Studio, Visual Studio 2012 supports remote debugging on x86, 64-bit, and ARM devices.

It may seem like an "in the weeds" topic for a first chapter, but many of you have Windows 8 ARM devices and may want to remotely debug the examples in this book using them. For that reason, I'll point you to the right resources here and also tell you what's possible with the tools.

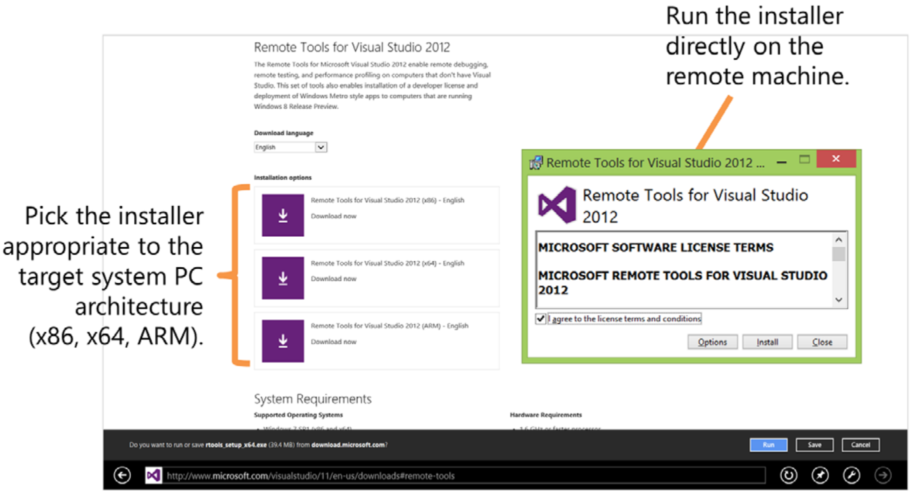
Before you can remotely debug on a machine, you have to install the remote components. Setting up remote debugging uses architecture-specific debugging components for the target machine. The remote debugging products are included with the DVDs for Visual Studio. But, because hardly anyone uses or keeps installation media, you can also download them from the Microsoft Download Center here: http://bit.ly/VS2012RemoteDebugInstall.

The setup and configuration have several steps that are machine-dependent. The ARM steps in particular are subject to change. Therefore, let me point you to the official source of information regarding the setup: http://bit.ly/Win8RemoteDebug.

You can debug only over a private network (home, work) or over a point-to-point Ethernet connection in public hard-wired between two machines. Debugging over the internet is not supported. Admittedly, it would be really cool to debug a machine halfway around the world across the public internet, like hackers do in movies ("This is Windows. I know Windows!"), but there may be one or two security concerns there. The connection also needs to be wired or wireless Ethernet, not USB, Serial, or another communications type.

You don't need to do this for the Simulator, or any other machine that already has Visual Studio 2012 on it (just run msvsmon directly from the Start screen), but the process for otherwise non-developer machines is as follows:

1. Install the remote components on the remote machine. You must do this as the administrator. Figure 1.9 shows the installer download page and install dialog running on the remote target machine.



Run the installer directly on the remote machine.

Pick the installer appropriate to the target system PC architecture (x86, x64, ARM).

Remote Tools for Visual Studio 2012

The Remote Tools for Microsoft Visual Studio 2012 enable remote debugging, remote testing, and performance profiling on computers that don't have Visual Studio. This set of tools also enables installation of a developer license and deployment of Windows Metro style apps to computers that are running Windows 8 Release Preview.

Download language: English

Do you want to run or save **rtools_setup_x64.exe** (39.4 MB) from **download.microsoft.com**?

http://www.microsoft.com/visualstudio/11/en-us/downloads#remote-tools

Installation options
Remote Tools for Visual Studio 2012 (x86) - English
Download now
Remote Tools for Visual Studio 2012 (x64) - English
Download now
Remote Tools for Visual Studio 2012 (ARM) - English
Download now
System Requirements
Supported Operating Systems
Hardware Requirements
Run
Save
Cancel
Remote Tools for Visual Studio 2012 ...
Remote Tools for Visual Studio 2012
MICROSOFT SOFTWARE LICENSE TERMS
MICROSOFT REMOTE TOOLS FOR VISUAL STUDIO 2012
☑ I agree to the license terms and conditions
Options
Install
Close

Figure 1.9 Installing the remote debugger on the remote target machine

2. Run the remote debugger on the remote machine. You must be an administrator to run this for the first time. Also, you'll need to obtain a developer license for the machine, just as you did for the main developer machine. After that, any regular user can start it as long as they're configured in the security dialog. Figure 1.10 shows where the remote debugger may be found on the target machine.

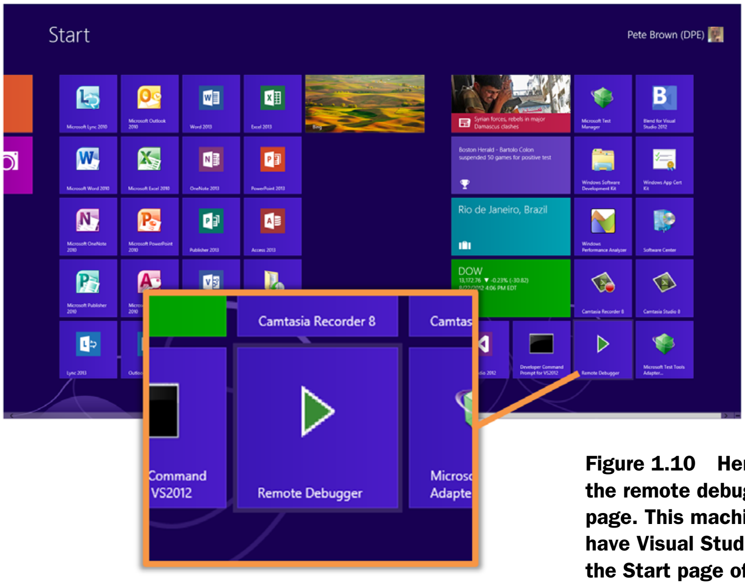
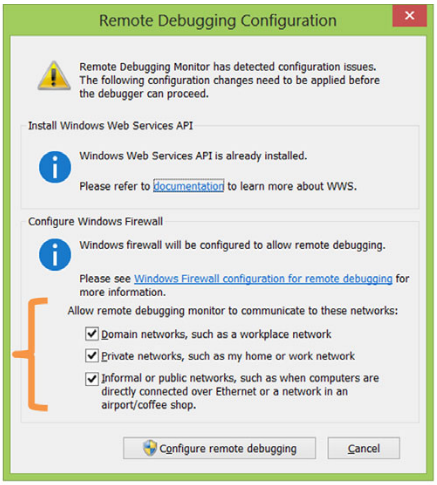


Figure 1.10 Here’s where you’ll find the remote debugger. It’s on the Start page. This machine happens to also have Visual Studio, but you’ll find it on the Start page of any regular device.

3 Configure the remote debugging so that it works through the firewall on the type of network you’re using (domain network, private network, ad hoc). Figure 1.11 shows the dialog where you make these settings. Be sure you’ve checked all of the options or the correct ones for your specific network. Sometimes what you think of as a private network is actually configured in Windows as a public network (turn on sharing to change this).


Remote Debugging Configuration

Remote Debugging Monitor has detected configuration issues. The following configuration changes need to be applied before the debugger can proceed.

Install Windows Web Services API

Windows Web Services API is already installed.

Please refer to documentation to learn more about WWS.

Configure Windows Firewall

Windows firewall will be configured to allow remote debugging.

Please see Windows Firewall configuration for remote debugging for more information.

Allow remote debugging monitor to communicate to these networks:

☑ Domain networks, such as a workplace network

☑ Private networks, such as my home or work network

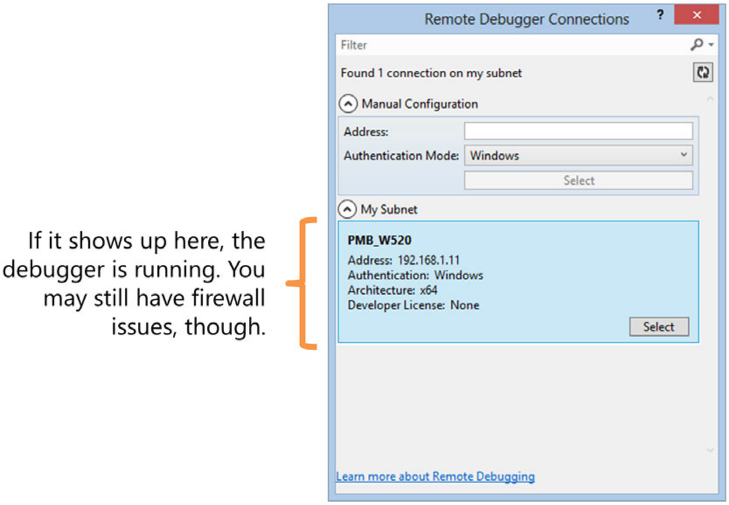
☑ Informal or public networks, such as when computers are directly connected over Ethernet or a network in an airport/coffee shop.

It’s critical that your network settings are correctly configured and allowed here.

Figure 1.11 Make sure you correctly set the network settings on the target device. If your network isn’t correctly recognized as private, but it is, click the network icon in the taskbar and right-click the connection and enable sharing.

4 Once the debugger is running, set the security method. You can turn off authentication if you're in a safe spot, but normally you'd keep the authentication at regular Windows authentication. This is done through the Options menu in the remote debugging monitor.

5 Set the Visual Studio debug target to be a remote machine. You'll be prompted for the machine to connect to. Make sure the debugger is running remotely before you do this. Figure 1.12 shows the Visual Studio dialog on the development machine.


Remote Debugger Connections
Filter
Manual Configuration
Select
My Subnet
Select
Learn more about Remote Debugging

Found 1 connection on my subnet

Address:

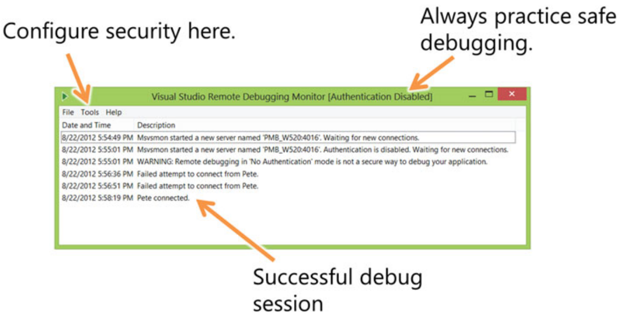
Authentication Mode: Windows

PMB_W520
Address: 192.168.1.11
Authentication: Windows
Architecture: x64
Developer License: None

If it shows up here, the debugger is running. You may still have firewall issues, though.

Figure 1.12 Configuring the remote debug target. If you need to reconfigure the debug target, you can find the info in the Debug tab of the project properties.

6 Start debugging as you normally would. If you get deployment errors where the device can't be connected, check your network settings. Depending on the configuration of the device, you may need to make the current connection a private network, or you may have authentication problems.


Visual Studio Remote Debugging Monitor [Authentication Disabled]
File Tools Help

```
Date and Time        Description
8/22/2012 5:54:49 PM Msvsmon started a new server named 'PMB_W520:4016'. Waiting for new connections.
8/22/2012 5:55:01 PM Msvsmon started a new server named 'PMB_W520:4016'. Authentication is disabled. Waiting for new connections.
8/22/2012 5:55:01 PM WARNING: Remote debugging in 'No Authentication' mode is not a secure way to debug your application.
8/22/2012 5:56:36 PM Failed attempt to connect from Pete.
8/22/2012 5:56:51 PM Failed attempt to connect from Pete.
8/22/2012 5:58:19 PM Pete connected.
```

Configure security here.

Always practice safe debugging.

Successful debug session

Figure 1.13 The Remote Debugging Monitor showing my debugging session

You can see a few things in figure 1.13 that were specific to my setup. One, I had it set to Authentication, but the domain server could not be contacted, so that failed. Then I changed it to No Authentication, but I forgot to check Allow Any User to Debug in the options dialog. Finally, I checked that option in the Tools > Options menu of the remote debugging monitor, and it worked.

> **NOTE** If you don't already have one, you will be prompted to get a developer license on the remote machine.

For ARM devices and other low-power tablets, remote debugging is essential; you can't run Visual Studio on everything. But even if you have two developer-class machines with only one display each (two laptops, for example), you may find debugging remotely to be more convenient than debugging on a lone single-display machine. Doing it this way avoids the context switching you have when bouncing between Visual Studio and the Windows app.

For scenarios where you just need to check different resolutions or simulate touch, using the Simulator is usually the best way to go. Be sure to try all three approaches—local, Simulator, and remote—to see which one best fits your workflow.

1.6 Summary

Before getting into the details of design and APIs, I wanted to start the book off with getting your hands dirty. In this very first app, you learned how to create a Windows 8 app from scratch using the Blank Application template.

Perhaps without realizing it, you used the layout grid to conform to the Windows 8 Modern design guidelines covered in the next chapter. You then used the application to connect to a service, download XML, and display it using a `ListView` with a `WrapGrid` and custom items templates. Not bad for a Hello World application!

For the majority of us, this was all run on the local machine. I prefer to debug on the main machine whenever possible, but when working with my ARM device, I need to use remote debugging, and when testing how the app will look at different resolutions, I use the Simulator. All of these together make it possible to test a large number of different configurations, all from your main development PC.

One of my favorite things about developing for Windows 8 is how it is so similar to other technologies I've written for in the past. Windows 8 XAML apps are written very much like Silverlight apps. If you have experience in the latter, you'll find yourself well equipped to move forward. If you don't, Silverlight was proven to be very easy to learn by anyone with basic C# skills, and I expect Windows 8 XAML to be just as easy and just as fun.

Personally, I'm looking forward to putting some apps out there. I hope you are too. In the next chapter, we'll take a look at the new Windows 8 design aesthetic and the UI conventions to give you a baseline to use when building your own apps.

2

The Modern UI

This chapter covers

- The Windows Modern Style
- Design principles for Windows 8 apps
- Typography and grid layout
- Device considerations

I'm not a designer. Like many, I can tell good design when I see it, but my promising art skills were tossed out the window and never really developed once I sat down at my first computer and started programming. Had I known then how well computers and art would coexist, I'd have kept pursuing them together.

Regardless, like many of you, I find myself designing UIs for applications on a fairly regular basis. In the days of rigid battleship-gray apps, this was relatively easy to do. As those fell out of favor and we started getting more creative, it became harder to keep up. Part of the reason it was hard to keep up was that there were few working constraints. There was no commonality to design, no framework to work within. Unlimited possibilities can be pretty daunting when you're not sure where to start.

With Windows Phone and Windows 8, Microsoft has attempted to bring us all back into the fold of a visual framework we can all understand. Those with design talent will still be able to create applications that outshine what the rest of us do, but all of us can now more effectively learn from each other and use the same tools and patterns to create applications. That framework is the Windows Modern Style.

> **METRO?** Metro was the code name for the design language. At Microsoft, we don't refer to apps as Metro-style apps but instead as Windows 8 apps or Modern apps. The aesthetic is simply referred to as the Windows design aesthetic, Windows Modern Style, Windows Store app design, or, more succinctly, the Windows style. In general, apps are referred to as Windows x and desktop applications and features are referred to as Windows desktop x.

The Windows Modern Style has many parts. First, I'll cover a bit of the inspiration for the language and its roots in past products. Then, I'll discuss the principles that govern the decisions about what makes something fit the aesthetic and whether or not to go a particular route in the design of your own application. These guiding principles will be a great help to designers everywhere. Next, I'll get into some of the more concrete aspects of Windows 8 app design: specifically, the importance of typography and its correct use and the idea of the layout grid. We'll wrap up this section with one of the main drivers for the new style: touch interaction. This entire chapter will be about the visual design for Windows 8 apps, what drives them, and how to fit into it. We're not going to discuss how to put a UI element on the page but rather why that UI element should even be there and how it should look.

2.1 Design inspiration

Although you can see elements of the Windows Modern Style in even earlier work such as Encarta and MSN, the design language has its closest implementation roots in the Zune client software for Windows. This client, written using the same pre-WPF presentation APIs as Media Center, introduced Windows users to the clean typography-centric design and borderless, chromeless windows. Figure 2.1 shows the Zune client, because I know it may be unfamiliar to many of you.

Silverlight for the desktop even had downloadable Zune-inspired navigation templates that followed the principles of its design at an application-level. Later, key elements of this design aesthetic, such as the tile approach to application launching and the use of text and case to distinguish purpose, were refined in Windows Phone 7. Now, this design aesthetic is an important part of Windows 8, Xbox 360, Windows Phone, and more.

At first, the Windows Modern Style may seem to be a radical new design that just popped out of the designers' brains at Microsoft. It appears fresh and interesting, yes, but like many great designs, it does it by borrowing very heavily from the excellent real-world design work that preceded it.

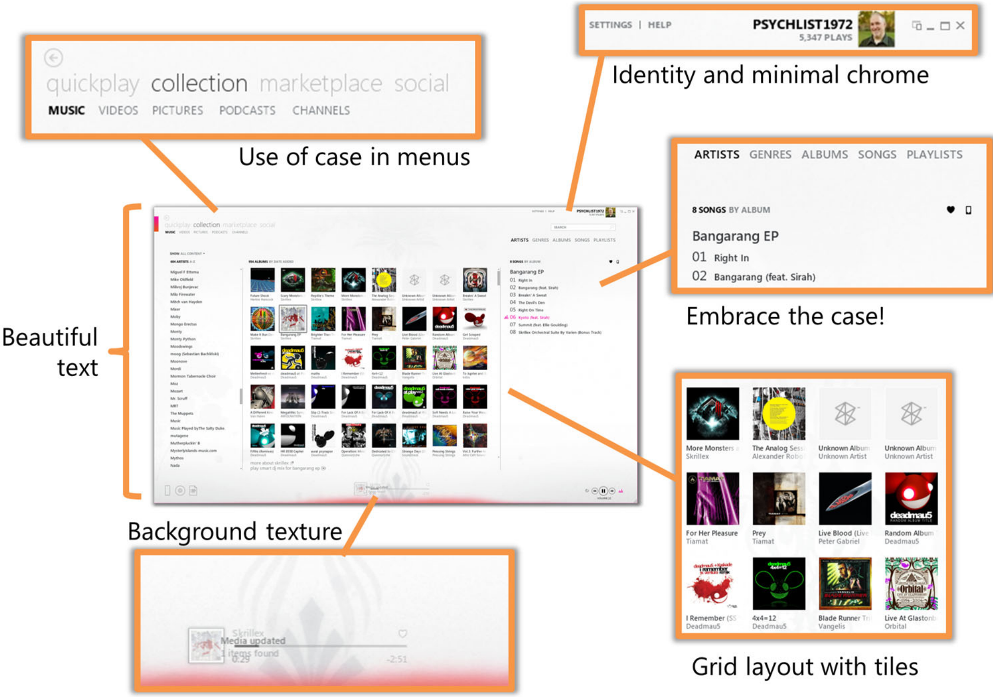

quickplay collection marketplace social
MUSIC VIDEOS PICTURES PODCASTS CHANNELS
Use of case in menus
SETTINGS | HELP
PSYCHLIST1972
5,347 PLAYS
Identity and minimal chrome
ARTISTS GENRES ALBUMS SONGS PLAYLISTS
8 SONGS BY ALBUM
Bangarang EP
01 Right In
02 Bangarang (feat. Sirah)
Embrace the case!
Beautiful text
Background texture
Media updated
Grid layout with tiles

Figure 2.1 I like my Zune client, and I've always liked its UI style; it implements an early version of what evolved to be the Windows Modern Style. You can clearly see the elements that continued forward into the design language used on Windows Phone and Windows 8.

2.1.1 Direct influences

The Windows Modern Style has three key design influences: modern (Bauhaus) design, the International Typographic Style also known as Swiss Design, and motion design as used in cinema. Each of those contributed key stylistic elements or concepts to complete the Windows 8 Modern design language.

- *Modern design* taught us to cut our designs down to the minimum required to meet the purpose while still being beautiful. Much like the American Craftsman movement in furniture was a reaction to the frilliness of Victorian design, modern design was about further removing excess adornment and simplifying design to its bare essentials, removing even the construction details that Craftsman showcased. In a house, some may see modern design as cold or hard, but in a computer interface, it's entirely appropriate and welcoming.
- *Swiss Design* taught us the importance of clean, crisp typography to convey information quickly. It showcased grid layout and bold, flat color. It was about conveying information quickly without requiring the mental effort of deciphering complex multicolor icons. It's beautifully stark, modern, and direct. The best

examples of this approach in the real world include the typography-centric wayfinding signs commonly used for public transportation systems, such as airports, bus terminals, and subways. The Seattle-Tacoma International Airport, which services the Microsoft main campus, is full of examples of this design. In those situations, simple, easily understood symbols and easily read text rule. Next time you're at an airport or in a subway, especially those in Europe or in large metro areas, look at the official signs and the approach used in their design. You'll see a little bit of the Windows 8 design aesthetic right there.

- *Motion design* taught us the importance of movement. It helps connect with our emotions as users. Some of the best examples of motion design include some movie opening credits, news story transitions and openers on TV, most video car ads, and more.[1] Certainly, however, motion design has been used in our world as well, in 3D graphics and animation; in Flash, Silverlight, and WPF; and more.

So the Windows Modern Style is as much an evolution as it is a revolution. It provides a new, modern UI approach but standardizes it so that everyone can once again be on board with using the same elements and styles across their applications. It takes its inspiration from real-world elements but makes them authentically digital.

2.1.2 Finding your way

Windows 8 ships with a number of apps, and many more are available in the app store. One of the built-in apps is the Photos app. This very simple app, shown in figure 2.2

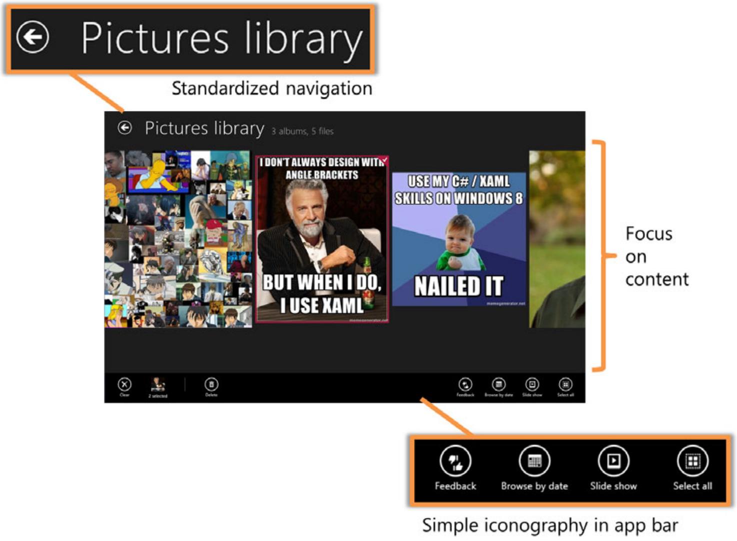

Pictures library
Standardized navigation
Pictures library 3 albums, 5 files
I DON'T ALWAYS DESIGN WITH ANGLE BRACKETS
BUT WHEN I DO, I USE XAML
USE MY C# / XAML SKILLS ON WINDOWS 8
NAILED IT
Focus on content
Feedback
Browse by date
Slide show
Select all
Simple iconography in app bar

Figure 2.2 The Windows 8 Photos Browser showing a couple of my favorite meme images. Notice the navigation arrow at the top left, the simple iconography, and the prominent use of typography in the title.

---

[1] One of my favorite blogs is "The Art of the Title" (www.artofthetitle.com/). Once you wrap up this chapter, pay them a visit and check out some of the introductory clips. Pretty much anything by Saul Bass is a great example.

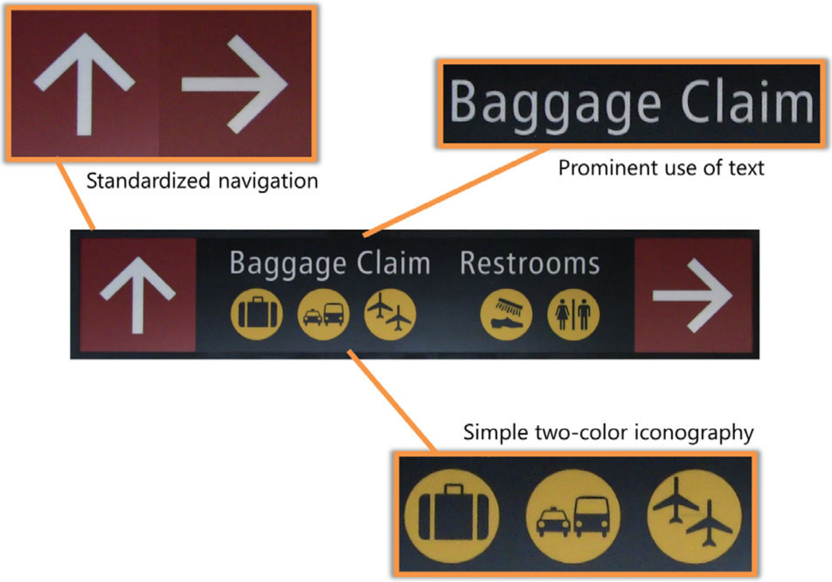


Figure 2.3 One of the inspirations for the Windows 8 aesthetic: way-finding signs at the Seattle-Tacoma International Airport. Notice the bold colors, simple icons, large type, and clear arrows. It's designed for quick scanning while you're moving from one area to another.

with the app bar open at the bottom after going into the pictures library and selecting a picture, shows many of the elements common to Windows 8 Modern apps.

On the top left, you can see the navigation arrow, and to its right, the application title. Below those, occupying the majority of the screen space, is the content. At the very bottom, you see the app bar. This isn't always visible; on a non-touch device, you right-click the space at the bottom to view it. On a touch screen, you swipe upward to make it visible.

The icons on the bottom are very simple with just two colors: the shape color and the background color. This simplicity, the navigation arrows, and the use of large text can be seen in airports and metro stations around the world. Figure 2.3 shows a photo I took at Seattle-Tacoma International Airport—the airport most local to Microsoft's headquarters in Redmond, Washington.

The similarities between the way-finding signs at the airport and the UI elements in Windows apps are, as you'd expect, quite numerous. It's all about quickly finding your way without having to spend the mental effort deciphering cryptic navigation elements.

Now that you know what the design looks like, you may be tempted to jump right into creating apps. But before using the style in your own applications, you'll need to understand the principles behind it.

2.2 Governing principles

In technical work, understanding the "why" of something is at least as important as understanding the "how." Similarly, when considering a new approach to visual design, it helps to understand the guiding or governing principles. These are the reasons the design is the way it is. In the case of the Windows Modern Style, there are five governing principles of design:

- *Take pride in craftsmanship.*

  Quality comes in at the beginning. Beautiful code, beautiful design. Sweat the details and do it the best you can from the very start. Do it well and be awesome. You want your apps to stand out in the store and bubble up as useful, well written, and well designed.

- *Be fast and fluid.*

  Be intuitive with motion, and delight the user. Be immersive and, perhaps most important, be responsive. You've heard "be fast and fluid" from Microsoft a million times,[2] but there's good reason for that. A choppy or slow UI hurts not just your application but the perception of the entire platform. I have a tablet that runs another tablet OS and found the stuttering in the UI, lag when dragging, and other lack of smoothness really get in the way of the experience. It's very obvious and viscerally disturbing when you drag something with your finger and it doesn't respond.

- *Be authentically digital.*

  Be connected to the rest of the world, to the cloud. Expect your user to be connected. Be dynamic and alive in your interaction. Use bold, vibrant colors. Use typography beautifully. Use motion to convey meaning. Don't pretend to be something you're not. Don't put a bookshelf in your app; don't make something look like the real-world version. This is exactly the opposite of the iPad design aesthetic, which goes for skeuomorphism,[3] or simulating real-world analogs. That simply doesn't fit the Windows aesthetic and will look as out of place as cheap veneer furniture in a gallery full of solid wood creations.

- *Do more with less.*

  Be focused and direct. Put content before chrome, information before extraneous design. Inspire confidence in your users. This is why the app bar, charms, and other chrome are normally hidden: You want to focus the user on the content of the app. They should be completely immersed in the goal they're trying to accomplish and not distracted by toolbars, icons, menus, and more.

- *Win as one.*

  Take advantage of the user's knowledge of the Windows 8 platform and work with the UI model, not against it. Work with other applications using contracts; don't feel that you need to invent everything yourself. Try not to invent new interaction patterns, but instead use those already present in the platform. When in doubt, use the tools and templates built into the design and development tools.

---

[2] It was a drinking game at //build 2011, even.

[3] See "Skeuomorph," http://en.wikipedia.org/wiki/Skeuomorph. No, it's not that beast from *Aliens*—that was a xenomorph.

These principles should serve as guidelines when creating your Windows apps. When in doubt about a UI element, go back and see if it meets these principles. If it violates one or more of them, consider looking at another way to accomplish the same functionality. The Build videos at http://buildwindows.com are great references here, as is the design-focused http://design.windows.com site.

Also, although I encourage you to use designers for every application you develop, the simplicity of the Windows Modern Style design language makes it approachable for design-aware developers to create great-looking and -performing, fast and fluid, authentically digital applications for Windows 8. Typography is a big part of the design and its simplicity.

2.3 Typography

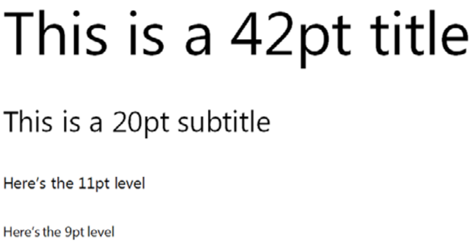
The Windows Modern Style is typography-centric. That is, it uses text for emphasis where we often used lines and other chrome elements in the past. In order for this to work, your type must show a clear hierarchy with significant point size differences between the different levels. For example, main headings should be in a very large font, and subheadings should be in a font size roughly half the size of the main heading. The next level down is roughly half of that size. The final text size is normal reading size.

For any given application, Windows apps generally use just four font sizes to establish the type hierarchy. The recommended font sizes are as shown in table 2.1.

Table 2.1 Recommended font sizes for Windows apps

Level	Point size
Level 1: Page headers (do not wrap this text)	42
Level 2: Subheaders and content headers	20
Level 3: Navigation, body copy, links, and more	11
Level 4: Tertiary information, field labels, and more	9

For each level, I've shown one point size that may apply. This is not a hard-and-fast rule; there's room to innovate and be creative. The primary rule is that there should be a very clear hierarchy so that the screen is easily parsed by the user and so that they can focus on the task you want them to perform. When in doubt, sticking with the established sizes will make your app look more natural, make it easier for you to work with the included templates, and take the guesswork out of setting up your hierarchy.



This is a 42pt title

This is a 20pt subtitle

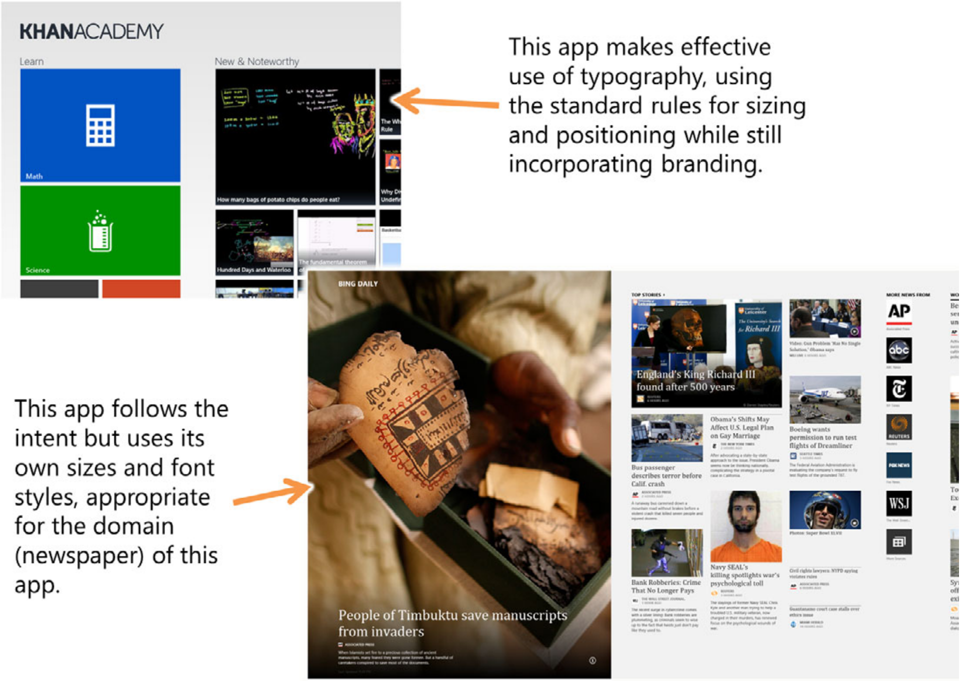
Here's the 11pt level

Here's the 9pt level

Figure 2.4 The default font hierarchy for Windows apps

Figure 2.4 shows the default type hierarchy. Notice how there's no ambiguity between levels one through three; the font sizes are significantly different.

The font used is typically a Segoe variant such as Segoe UI. When in doubt, use the fonts and font sizes provided in the default templates. When you need to, you can certainly vary from those fonts: Windows supports beautiful OpenType text, as you'll see in chapter 8. But keep in mind that doing so should be reserved for times when you want a domain-specific (such as in games or news) or brand-specific look to your application. Figure 2.5 shows two cases. The first, Khan Academy, uses a branded logo but sticks with the default font hierarchy otherwise. The second, Bing News, uses a domain-specific font and layout, eschewing the defaults. It retains a clear type hierarchy, however. Even looking at the image in this figure, you should be able to see which lines are headlines versus body text and where the lead story and its title are located.


KHANACADEMY
Learn
New & Noteworthy
Math
Science
How many bags of potato chips do people eat?
Hundred Days and Waterloo
The fundamental theorem
BING DAILY
People of Timbuktu save manuscripts from invaders
TOP STORIES
England's King Richard III found after 500 years
MORE NEWS FROM
AP
abc
WSJ

This app makes effective use of typography, using the standard rules for sizing and positioning while still incorporating branding.

This app follows the intent but uses its own sizes and font styles, appropriate for the domain (newspaper) of this app.

Figure 2.5 Two examples of typography that doesn't follow the letter of the law but certainly follows the intent. Khan Academy is very close, except it uses a branded header (which is a good idea). The Bing News app is very readable and maintains a strong hierarchy but uses domain-specific layout and type styles. Both are perfectly valid approaches for Windows Store apps.

When choosing the font sizes, the Windows design team worked with the concept of a master layout grid. The sizes work well with the grid, aligning properly at different levels.

2.4 The importance of the layout grid

Remember the old Windows style guide and how it specified exact pixel measurements between elements in a dialog? Remember "twips" and "dialog units"? Back in the '90s these guidelines were considered the absolute last word on UI design in Windows. Later, when UI design evolved, much of this was tossed out the window, and many Windows UI Design Guideline books became doorstops and monitor stands or, at best, collector's items.

Windows 8 brings us back to grid layout with well-defined spacing. All UI elements in a Modern Style app are laid out on a 20-pixel grid. Each 20-pixel square is called a *unit*. The grid is further subdivided into 5 x 5–pixel squares to assist with spacing of elements.

Looking back to typography, for a standard app the baseline of the first line of typography, the heading, is on a unit line, five units from the top (5 * 20 pixels). The content is at the top of the seventh unit from the top (7 * 20 pixels).

Furthermore, the baseline of text at the 11- or 15-point size should sit on a 20-pixel unit line. Figure 2.6 shows the grid in action with an example heading in place.

The built-in Visual Studio and Blend templates make aligning to the grid simple by default. Unlike the approach with Silverlight (and WPF), the project templates all follow the same layout guidelines and grid pattern. Even the "blank" project template comes with all the required assets and guides to make it easy for you to build a Windows-style app that adheres to the standards. Figure 2.7 shows an example.

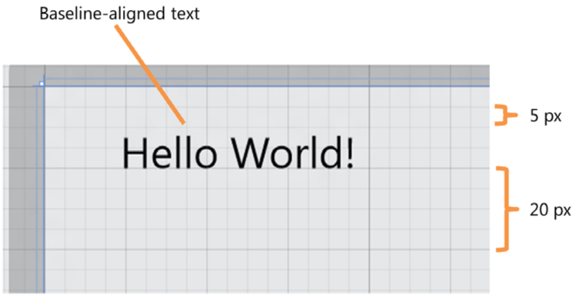


Figure 2.6 The layout grid, not to be confused with the `Grid` layout control in XAML

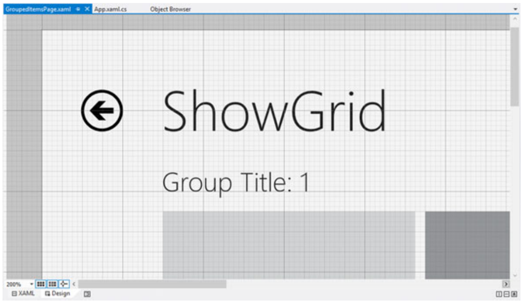


Figure 2.7 The default templates make use of the grid for layout and alignment. Note how elements always start on grid lines and how the text baseline and button center are all grid aligned. The smallest grid division shown is 5 px, the next is 20 px, and the boldest grid lines are 80 px.

I'll cover the templates and the grid more as we create a number of different projects throughout the remainder of this book.

One final important aspect of the Windows Modern Style is the importance of touch for manipulation.

2.5 Design for touch but not only for touch

The design aesthetic, whether it is on the phone or the tablet, is designed for touch-first experiences. Keyboard and mouse are supported, of course, but touch is now a first-class citizen of the UI manipulation world. What does it mean to design for touch versus designing just for the mouse? A pointer is a pointer, right? Not exactly.

First, there's the physical issue of accuracy. Even on a high-DPI display, a mouse pointer can easily be accurate down to a pixel or two on the screen. Your finger is a much larger physical device. Not only is it less accurate, but the very act of using it often obscures what's displayed on the screen. This is driven home for me every time I try to select text on my phone—it's like performing brain surgery with garden tools.

Second, a finger is direct manipulation rather than indirect manipulation. You don't need to move a surrogate pointer around the screen to get to where you want to go; you simply pick up your finger and then put it down on the target spot. This tends to make movement more accurate even when actual pixel-level targets are less so. This also means that mouse-centric UI notions like "hover" aren't appropriate in a touch world.

Third, you likely have 10 fingers (or at least some number greater than two depending on how careful you've been with that table saw), whereas it's unlikely you have more than one active mouse. Multiple touch points mean that the device can start to use multipoint gestures in place of multistep mouse manipulations. Consider the task of zooming in on an image. A common way to do this on the mouse is to either use the mouse wheel or right-click. With touch, the accepted gesture is to "pinch" to zoom out and move your fingers apart to zoom in. Your fingers have no built-in buttons or scroll wheels, so gestures are very important.

Design for touch, but remember that, at least on Windows 8, touch isn't the only way to navigate. You'll also need to support the mouse and, if you're thorough, keyboard navigation and manipulation as well.

The principles of the Windows Modern Style design aesthetic apply to a number of different platforms produced by Microsoft. On Windows 8, it's more than just the style or design language; it's an entire class of application.

2.6 Modern apps on Windows 8

So far, we've covered the Windows Modern Style rather broadly but with an eye toward Windows Store apps. The design language is broader than Windows 8 and the Windows Store, but in Windows, an "app" implies more than just the design language. It implies a type of application using a specific set of APIs. Technically, these applications are Windows Runtime Modern Style apps, but they're more commonly referred to simply as Windows apps or Windows 8 apps.

Windows apps can be very broadly classified into consumer applications and enterprise line-of-business applications. There are lots of gray areas in between, but in this case, the classification depends on whether you distribute your app through the Windows Store or through enterprise sideloading or an enterprise app store.

Regardless of how you classify the application, a number of important or key user interface elements will be used. For example, apps will almost always have an app bar (the fly-out toolbars at the top and bottom of the screen) and will certainly have access to charms (the fly-out toolbar on the right), even if the app code doesn't explicitly integrate with any of them. Each of these key elements will be covered in this section so that you can readily identify them in applications you try out.

Finally, apps on Windows 8 can target a large variety of form factors, even more now than we had available to us in the past. Your app may be intended for tablets, but it will have to work on laptops and desktop machines, and vice versa. This can be a pain for testing, but it's one thing that makes the Windows platform so special: choice.

Each of these areas includes important aspects that must be considered when building Windows 8 Modern Style apps. Let's start with the difference between consumer and enterprise apps.

2.6.1 Consumer and enterprise apps

Apps are fundamentally different from their desktop application counterparts. You've already learned about the importance of touch, typography, simplicity, and the grid. They need to load faster. They need to be less feature dense. They need to be authentic, and they should be beautiful.

Beyond that, most apps, at least at first, will be consumer-oriented applications sold through the Windows Store. But Windows 8 apps aren't limited to just consumer apps. Enterprise line-of-business applications are also supported. Except for the distribution method, the dividing line between what makes a consumer app and what makes an enterprise app can be very gray indeed.

CONSUMER APPS

Most of the interest around building Windows apps is in the consumer space. Consumer-style applications share a number of common traits:

- *Small in size*—These apps are often downloaded over relatively slow connections. They need to be small and easily (and quickly) acquired. Small download times and fast startup times are almost always related.
- *Small in scope*—Consumer applications often do a very small number of things and do them well. Rather than have giant applications that do everything in a particular domain, each app does a subset of the features. Consider the typical calendar, email, and task-management functionality. These are often bundled into large desktop applications. On Windows 8, you'll almost certainly see them as three different applications that can communicate with each other. This lowers the learning curve and keeps the UI simple.

- *Secure*—Consumer apps need to be trusted. The consumer needs to believe that if they download an app from a trusted source such as the Windows Store, the app can't do nasty things with their machine or compromise privacy.
- *Self-contained*—Consumer apps typically provide value without relying on other applications installed on the machine. In fact, your app won't pass Windows Store acceptance criteria if you require certain desktop apps or even other Windows apps to be installed first. In other words, there's no COM automation of apps or trying to open a socket to speak to a desktop app running a service.
- *Installed from a trusted store*—You don't simply go to a random site and download apps for your machine. Instead, these apps are purchased through the store functionality in the OS itself. This makes it easy to find and purchase applications and, more importantly, provides a trusted location for app purchases.

One thing I didn't mention is "inexpensive." Why? Although the iPhone is commonly credited for starting the trend of $0.99 apps, there's no reason why your apps have to be that inexpensive. Price points will be decided by the usual factors that come into play in an open marketplace. That said, downloadable consumer apps are almost always a fraction of the cost of big box store-packaged applications.

When combined with the new design principles and the touch-centric design, you can see that a Windows 8 app is definitely a different beast from what you've been developing for the desktop. If you've been writing for Android or iOS, you have a bit of a leg up in terms of deciding how to scope your applications. The modern equivalent of big fat desktop suites just isn't going to cut it in Windows 8; those are better kept on the desktop.

ENTERPRISE AND LINE-OF-BUSINESS APPLICATIONS

Corporate developers, don't feel left out! Windows 8, like the Windows Phone, certainly feels targeted toward consumer applications, but enterprise developers have a way to get applications on machines without releasing them to the public Windows Store. Domain-joined enterprise developers can use enterprise sideloading and code-signing keys to get the applications deployed within the enterprise.

One nice thing about enterprise sideloading is it enables you to access potentially more functionality than sandboxed store applications do. The reason is the Windows Store is the gatekeeper that checks to see if you're using APIs outside of the accepted surface area. Using additional functionality in this way is not currently a recommended scenario, nor is it one that's currently guaranteed to work in the future, but it's worth considering.

TIP Whenever possible, stick to Windows Store guidelines, even when developing internal apps. This will give you the most flexibility and greatest support going forward. If you must step outside Store guidelines for an enterprise side-loaded app, encapsulate the offending code as much as possible.

Does this mean that Windows 8 app development is enterprise-ready? That's still to be determined by you. What it does mean is that if you can effectively work with the set of APIs you have access to, you have a mechanism by which you can deploy these applications inside a company.

In any case, you'll still want to adhere to the principles of Windows Modern design and also keep your applications relatively small. Don't be tempted to create the mother of all portals as an app.

In Windows 8, apps share a common set of UI navigation and manipulation standards. Those are the key non-content interaction elements for the applications.

2.6.2 Key Windows 8 UI elements and states

UI standards that cover only pixel positioning and font sizes are nice for helping you with the visuals. They're just not helpful by themselves; you really need standards covering the experience and actual interaction in applications. Windows 8 has several standard onscreen elements and application states.

- *The app bar*—The app bar is the replacement for menus and toolbars in Windows 8 apps. It appears at the bottom and/or the top of the application in response to a finger swipe or a mouse right-click. In keeping with the content-centric application to application design, the app bar is normally hidden and out of your way. Your application controls what shows up on the app bar. It can contain anything from simple buttons to more complex filtering and sorting elements. We'll cover how to create your own app bar functionality in chapter 11.
- *Charms*—The charms bar is conceptually similar to the app bar but isn't controllable by your application. The charms appear from the right of the screen in response to a swipe from that side or a mouse click in the right corners. It is here that you'll find the entrance to your app's settings panel, search and share functionality, integration with devices, and the ability to navigate back to the main Start screen.
- *Snapped states*—This is not so much a UI element as it is a state your app can be in. If the screen is at least 1366 pixels wide, you can snap, or dock, apps on either the left or right of the screen. The main app will be at least 1024 x 768, and the smaller snapped application will take up the remaining horizontal space—320 pixels. A good application needs to provide an appropriate UI for both the normal state and the snapped state. This is covered in more detail in chapter 13.
- *The tile*—The app's tile on the Start screen is more than just a replacement for the old icons we've become used to. In addition to being a simple launching point, the tile can be used to convey ongoing information to the user. It's what all users will see whenever they start up Windows or go to open other apps, so it's a high-visibility area. The app tile is covered in chapter 12.

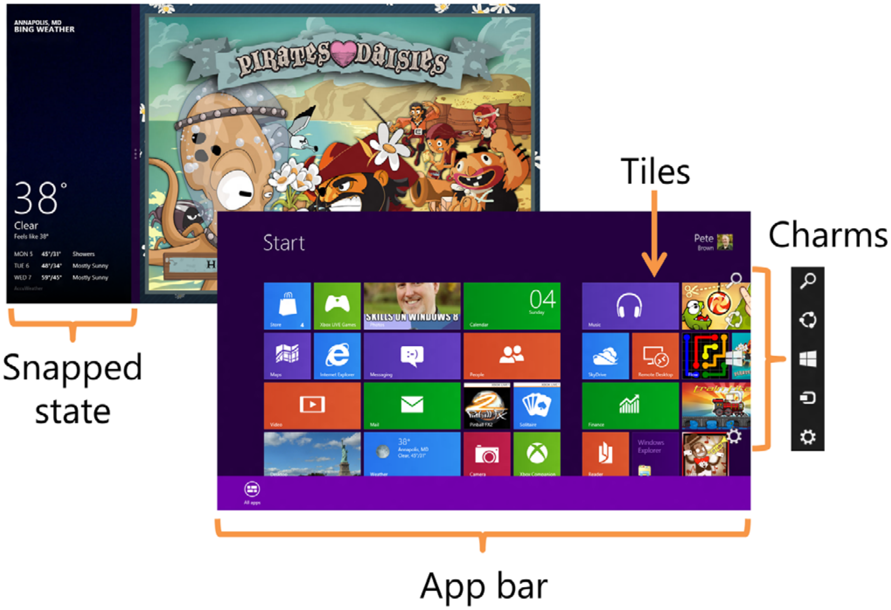


Figure 2.8 The key elements of a Windows app, shown on the Start screen and two snapped applications. Most apps aren't as visually busy as the Start screen, so you'll find that the charms show up better in those.

Inside your application, you can do just about anything you want. But you'll want to make use of the key elements and states. Figure 2.8 shows each of these elements.

> **ABOUT PIXELS** Not everything is as it seems. I mentioned that the snapped state is a fixed 320 pixels. But what about on super-high-resolution screens? If the DPI is above a certain threshold (determined by Windows), your screen will kick into a high DPI mode on install. In that mode, app pixels are not 1:1 with device pixels—you're working with device-independent pixels. That means that you code to a 320-pixel width, but the actual physical pixel width may be something like 480 or more. The result is much crisper text and finer lines with less aliasing. I'll dive into this more deeply when I cover the states in detail.

The snapped state is an interesting aspect of Windows design that is well suited to XAML. The snapped Bing Weather app shows a different UI in this view versus when it has the whole screen to itself. The remaining space on the screen is dedicated to the Pirates Love Daisies application. The mode Pirates Love Daisies is in is called "filled" because it "fills" the remaining space. When working with snapped views in XAML, you can scale the contents, wrap them differently, or simply swap out a different display, all using visual states. I'll cover that in more detail in chapter 12.

The snapped state brings out an interesting design consideration related to screen resolution. It only works if you have a screen width of at least 1366 pixels. Below that, snapping is disabled in order to always provide at least 1024 x 768 for the main application, the minimum required resolution for a Modern Windows app. Resolution is a very important device consideration, but it isn't the only one.

2.7 Device considerations

Prior to Windows 8, when you deployed an application to a Windows machine, you could be fairly certain you were running on a desktop or a laptop. Windows 8 significantly changes this by expanding the types of devices it actively targets. Sure, Windows 7 worked on tablets, and even Windows XP had a tablet edition, but those were niche products and were really just pen-operated desktop systems. As the interface for purely touch devices, they weren't successful.

With Windows 8, it's expected that many, if not most, consumer apps will be run from low-power ARM devices like tablets. As developers, we'll generally gravitate toward the beefier x86/64 machines, but consumers will buy primarily based on cost, weight, and battery life: all strong points for ARM devices.

So, in the Windows 8 world there are two primary form factors you'll need to work with: desktop or laptop and tablet or hybrid.

2.7.1 Desktop or laptop

Supporting desktops and laptops requires very little extra work on your part. For the most part, you'll use the same types of mouse and keyboard interaction as you do today. Although you shouldn't assume so, it is unlikely that any given desktop will have a touch screen unless you're targeting a specific type of kiosk, like a kitchen PC.

With a desktop you can typically count on a keyboard and mouse or some sort of indirect pointing device. You're also likely to see much higher resolution than on a tablet. I have two 30" displays each running 2560 x 1600. That's quite a bit more than the 1366 x 768 tablet spec, although they're relatively low DPI in comparison.

Desktop machines are also more likely than tablets or netbooks to have more than one display. In those cases, Windows apps will display on one monitor and the desktop on the others (or desktop on all, but we'll limit ourselves to apps here).

Finally, desktops and laptops are unlikely to have the rich set of sensors that many tablets have.

2.7.2 Tablet and smaller devices

Tablets differ from laptops and desktops not only in the obvious matters of form (screen resolution, touch screens, low power) but also in usage patterns. If not dedicated to a specific business use like field reporting for an insurance adjuster, tablets are typically casual-use devices. Your user is not likely to sit in front of the machine for hours on end or leave it running all the time. Instead, it will be quickly turned on, some activity performed, and then turned off. Often, they are used on the couch or on the train or in the back of a school bus.

For those reasons, you'll need to make sure your apps are fast and intuitive. If the user can't quickly perform their task without help, your app won't be used. Users rarely use help on the desktop, and they just about never use help on a tablet. F1 is as good as dead, ladies and gentlemen.

You'll also need to allow your application's sounds (if any) to be easily muted or any background music turned off.

In the Windows Store app world, you'll need to make sure your applications work well with touch, mouse, and keyboard. You'll need them to work on desktops without touch screens, desktops and laptops with touch screens, and tablets. Your applications will need to make use of the built-in Modern elements provided by Windows. All of these things are appropriate considerations for Windows Modern Style apps in the Windows Store, and most still apply to enterprise sideload applications.

2.7.3 Hybrid devices

Many of the devices for Windows 8 are hybrid devices. That is, they are part laptop or Ultrabook and part tablet. Even the Microsoft Surface includes a keyboard cover with a touch pad for pointer interaction. Some others are convertibles, and some like my Lenovo X220 or my wife's Acer S7 are simply laptops or Ultrabooks with a touch screen.

The usage patterns for these devices is different from both laptops and tablets. For example, on my Lenovo, I find that I often swipe with the screen to navigate and pull up app bars and the charms, but I do a lot of other interaction using the touch pad and keyboard.

All these form factors may seem like a huge burden, but as you'll see throughout this book, the tools and templates available to us, plus the power of the Windows Runtime (next chapter), make developing and designing these apps as easy as designing any other Windows application.

2.8 Summary

The Windows Modern Style design aesthetic emphasizes beautiful design that's authentic to the platform. It focuses on content over chrome and text over adornments. The style provides a framework to use when designing your own apps: recommended font faces, the grid layout, the type hierarchy, and more. Although you'll always benefit from including a designer on the team, the framework provides enough guidance that a developer with some design talent has a good shot at creating something beautiful and functional.

If I had to sum up the Windows Modern Style, I'd say the aesthetic is about getting things done while delighting the user. But it's not artificial delight like that imitation strawberry flavor in your milk; it's about delighting the user because the app is simple to use, easy to navigate, and easy on the eyes.

This chapter was about the design side of the equation. In the next chapter, we'll look at the development side of the equation, including how the Windows Runtime and .NET fit together to form the app development platform.

3

The Windows Runtime and .NET

This chapter covers

- Windows Store app system architecture
- The Windows Runtime
- .NET 4.5

At the start of 2000, after everyone stopped panicking about the impending Y2K doom, I got hold of some of the first alpha builds of what would eventually be .NET 1.0. I was part of a group that went around delivering two days of training on the upcoming .NET. At first, there was no IDE, and the bits were for building ASP.NET (or ASP Plus) pages. Nevertheless, to this VB6 programmer, it was clearly revolutionary, especially the brand-new C# language and the actual library of usable classes. (Remember, VB6 had no base class library.)

Six and a half years later, I got some of the first bits for Silverlight and went on to develop the first deployed managed Silverlight app ever—a carbon calculator written in Silverlight 1.1 alpha. Silverlight seemed as revolutionary to me as .NET did more than half a decade earlier.

Now, six years after that, we have another revolutionary platform—the Windows Runtime (WinRT). WinRT may look similar to .NET, but it's unique in its own way, building on the successes of .NET and Silverlight, plus some secret sauce from the C++ and JavaScript teams. As you'll learn, WinRT isn't a replacement for .NET, but it does take care of much of the heavy lifting that used to be done in managed code.

In this chapter, I'll introduce you to WinRT and .NET 4.5. These are the underlying APIs you'll use for building all Windows Store (Modern Style) XAML apps. We'll first take a look at what WinRT is and how it relates to .NET and .NET metadata. As part of that, you'll learn about projections, the language-specific wrappers for WinRT. Then you'll learn a bit about the application model and sandbox that WinRT was built to support. Finally, we'll take a brief tour of the WinRT namespaces so you have a general idea of where to look for specific types of APIs.

In order to do all this, you need to understand what WinRT is and what it isn't, so let's start with a look of the overall system architecture that supports Windows Modern Style apps.

3.1 Windows Store app system architecture

In the past, the platform and the API we used were separate. VB3-6 programmers had to wait for someone to wrap native Win32 (or COM) code to make the latest toolbar or UI widget accessible and friendly to the programming language. Similarly, .NET developers often had to wait for the same type of wrapping to happen. This meant that Windows would come out with great new features, but it would often be at least a version or two before mainstream application developers were able to use it.

This was frustrating for developers, for users, and for the Windows teams themselves. I mean, why create a new API when only a small percentage of your audience will be able to use it?

With Windows 8, the Windows team took ownership of the API and programming model. Not only did they provide a modern and programmer-friendly API, but they also provided a host of other components to make for a complete development platform. Windows Store apps build on WinRT and all the other support provided by the system. Just as a Win32 app builds on more than just the Win32 API, Windows Store apps build on more than just WinRT.

Figure 3.1 provides a high-level look at the overall system architecture. A fairly large number of pieces come into play in Windows Modern-style apps. Here are a few of the more important ones:

- *The Windows Store app* is the app you're building. Throughout this book, that will be a C# .NET 4.5 Modern Style app.
- *The language projection* provides a language-friendly interface to WinRT. More on that in the next section.
- *The Windows metadata, namespace, all the sub-namespaces, and the core* together make up the Windows Runtime. This chapter, and much of this book, deals

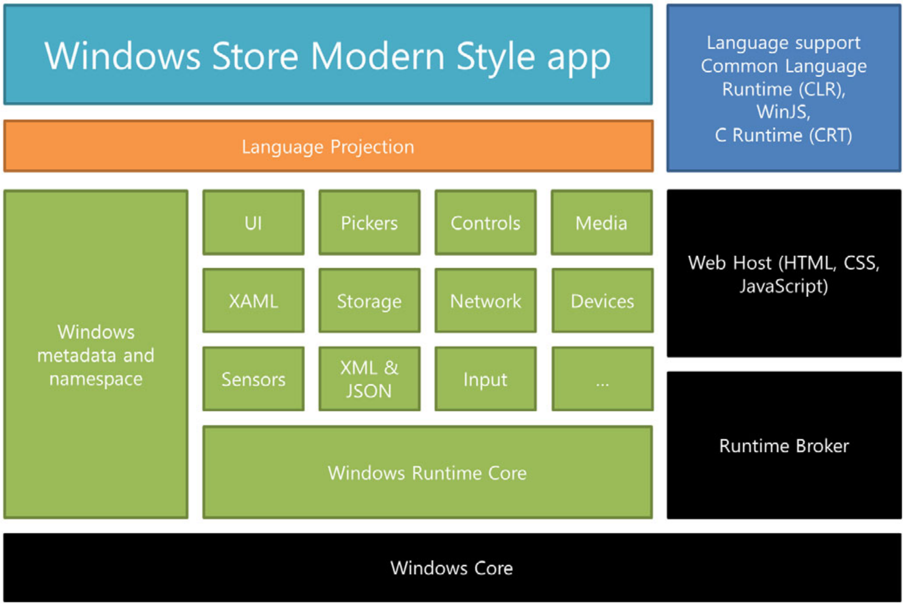

Windows Store Modern Style app
Language support Common Language Runtime (CLR), WinJS, C Runtime (CRT)
Language Projection
Windows metadata and namespace
UI
Pickers
Controls
Media
XAML
Storage
Network
Devices
Sensors
XML & JSON
Input
...
Windows Runtime Core
Web Host (HTML, CSS, JavaScript)
Runtime Broker
Windows Core

Figure 3.1 The architecture for Windows Store apps built on WinRT and one of the supported languages

specifically with WinRT. It's interesting to note just how much is there, however. All of the XAML stack is implemented as part of WinRT, in native code. So are the sensor access code, many core UI concepts, all the built-in XAML controls, network access, and much more. When browsing, you'll find this all in the Windows.* namespaces.

- *The language support* provides all the extra stuff for specific languages and platforms. On the .NET side, it's the Common Language Runtime (CLR) and the Base Class Library (BCL) for .NET 4.5. In .NET, you'll find this all in System.*.
- *The web host* provides a safe environment within which to run WinJS applications. Similar to what we had with Silverlight out-of-browser apps, it's the IE10 browser without the chrome and with the unnecessary stuff removed.
- *The runtime broker* handles access to things like device drivers and authentication. Unlike desktop apps, you don't automatically have access to every device and driver on the system; instead, the driver must be in a known class with a WinRT API (like webcam/microphone), or if a custom device, the driver must grant explicit permission to the specific device app in its manifest.
- *The Windows core* includes the Windows kernel, native function calls, DirectX, and more. It's the low-level stuff that public APIs build on. Because WinRT builds on this, it provides the shortest route from your code to the hardware, regardless of processor architecture.

Just as with any other development platform, there are other bits that come into play, such as Azure and other online services, third-party components and tools, and more. But the core pieces are all owned by the people implementing the core functionality. No more lag waiting for one team to wrap the features written by another! It also means that these features can all work together because the teams have more intimate knowledge of how they should work and perform. This may seem a no-brainer to folks on the outside, but anyone who has watched Microsoft (or worked there) knows that it wasn't easy getting everyone on the same train.

One key service not shown here is the Windows Store. That's not part of the underlying platform but instead supports Modern Style apps and serves as the gatekeeper for enforcing the sandbox.

In this section, I'll introduce the sandbox and briefly cover app packaging and the Windows app store, and I'll wrap up with a look at the driver model. More details for each of these will be included in the remaining chapters in this book, but it helps to have a taste of them up front.

3.1.1 The sandbox

For an application to be safe and to be trusted by users, it must run in some sort of a sandbox. Much like its real-life preschool counterpart, the app sandbox is a safe place for apps to play where they won't get hurt by outside interference and where they can't reach out and cause trouble. Beyond that, the analogy breaks down quickly: For example, you can't kick sand at other apps or throw toys at the kids on the swing set.

The main point of an app sandbox is to protect the user. This is enforced in four ways:

- WinRT exposes only safe APIs.
- The .NET 4.5 subset for Windows Store apps exposes only safe APIs.
- The code runs in a low-access, isolated space, sometimes referred to as the LowBox.
- The Windows app store verification tools perform a static analysis and make sure your app isn't accessing anything it shouldn't.

The first two points are important and help keep you in the right place by only exposing APIs you can use. The third helps with anything that runs wild and otherwise would escape. But it's the last point that really helps you figure out if your app is playing nicely in the sandbox. For example, you can p-invoke (call Win32 DLLs) from a Windows Store C# app all you want. But if you run that through the store verification process, it will almost certainly fail unless you stick to a documented set of safe Win32 or COM APIs.

What is the sandbox, anyway?

Each app you create can request capabilities by declaring them in the manifest. You'll learn about those throughout this book, but understand that they control access to resources such as the webcam and microphone, the documents library, intranet and internet connections, and the like.

These capabilities and the rest of the sandbox are enforced at the OS level through a new construct called an AppContainer.

The AppContainer is an integrity-level feature of the Windows 8 OS itself, not of WinRT, .NET, or XAML apps. It comes into play not only for WinRT apps but also for Internet Explorer 64-bit tabs. The Windows Store part of IE10 is 64-bit on 64-bit machines. In IE10 on the Intel 64-bit desktop, all tabs are 32-bit by default for backward compatibility with plug-ins. Enabling 64-bit tabs requires enabling Enhanced Protection Mode (EPM) through the browser properties.

The AppContainer model is similar to sandboxing models used in mobile operating systems, and it provides a much stronger app-centric approach to security. The intent, of course, is to avoid viruses and malicious software and to put the end user in control of what apps are allowed to do.

3.1.2 Deployment and the Windows Store

Any individual application destined for store distribution is packaged into an app package with the .appx extension, commonly referred to as "an app ex." App packages follow the Open Packing Conventions (OPC) standard, which means they're simply zip files that contain all the application binaries, content, and manifest files required to identify the application. If you have Silverlight experience, you'll find them very similar to a .xap file.

One key difference from a Silverlight .xap, however, is that during development, the .appx isn't automatically created. Instead, the application is run directly from the filesystem. You'll create the .appx only when you're ready to distribute it to others. It's also important to note that you can't simply send someone else an .appx and expect them to be able to install the app. The only options are these:

- Compile/build the app and run it from the same machine. This is the develop/debug scenario.
- Send it to the store so anyone can install and run it. For most people, this is the goal.
- Use enterprise sideloading if you're on a domain. Enterprise applications can be loaded without going through the store.
- Use an enterprise app store.

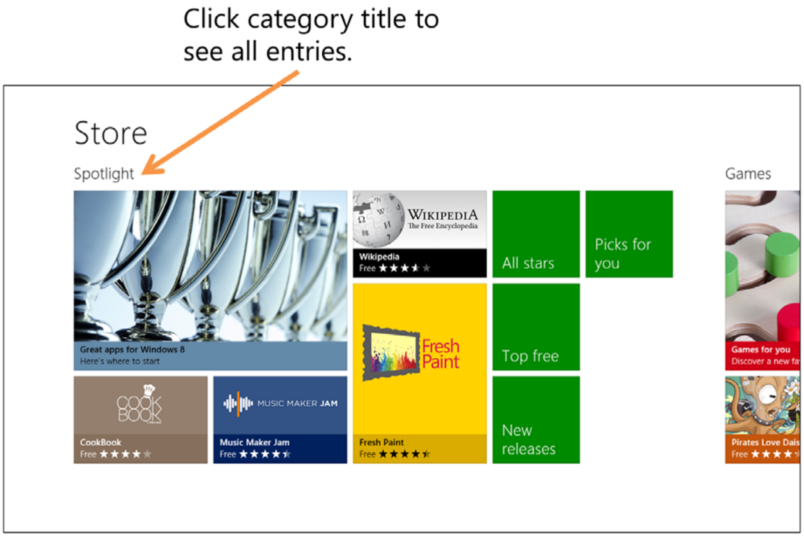

Click category title to see all entries.
Store
Spotlight
Games
Great apps for Windows 8
Here's where to start
Wikipedia
The Free Encyclopedia
Wikipedia
Free
All stars
Picks for you
Fresh Paint
Top free
New releases
COOK BOOK
CookBook
Free
MUSIC MAKER JAM
Music Maker Jam
Free
Fresh Paint
Free
Games for you
Discover a new fa
Pirates Love Dais
Free

Figure 3.2 The Windows app store. This is the primary means of deployment for Windows Modern Style apps.

Almost all consumer applications will end up in the public Windows Stores. That's how you get broad reach—and maybe make a little money. Figure 3.2 shows the store's main page. One often-missed detail of the store interface is that you can click the category heading to see all the entries in that category—this is typical grouped `GridView` behavior, as you'll see in later chapters.

I mentioned earlier that the store verification tools are what enforce the sandbox. We'll cover those, .appx packages, and deploying to the Windows app store in chapter 23.

Finally, the new Windows Store app model provides a new device driver model.

3.1.3 The driver model

In desktop apps, you can access any installed driver you can figure out how to use. Windows Store apps use a new driver access model that restricts device access. If the device doesn't fit one of the known types with an exposed WinRT API (like a webcam or microphone or network adapter) or allowed class driver, then access is restricted to the device app identified by the independent hardware vendor (IHV) or original equipment manufacturer (OEM) as part of the driver manifest.

Access to these devices is done through an API provided by the IHV or OEM, which works with the broker to provide access to the device itself, but only to the identified device app. This means that, currently, IHVs and OEMs can't provide drivers or APIs that can be used by any Windows Modern Style app. I expect this restriction to loosen in the future or for Microsoft to provide a different way for IHVs and OEMs to create devices with APIs that any app can use. I'm personally waiting for MIDI devices and will have to stick with something like Open Sound Control (OSC) over Wi-Fi until then.

These restrictions are primarily for security and stability reasons. A malicious device driver can be a pretty big attack vector. Not only that, but bad device drivers are responsible for most Windows crashes today.

The platform for Windows Store Modern Style apps in Windows 8 provides a lot of what we'd expect a full OS to provide. In some ways, it reminds me of how Windows 3.1 ran over DOS. It's quite conceivable that Windows Modern Style apps will be the primary way to build apps in the future, just as WinAPI apps became the primary way to build apps in Windows 3.1 and Windows 95.

Now that you see the larger picture and have a basic understanding of the major pieces in the platform, let's dive into the boxes in the middle: WinRT.

3.2 COM + .NET metadata = WinRT

WinRT is not .NET. Full stop. It looks like .NET. It seems to behave like .NET. Heck, it shows up in the object browser just like .NET. But no, it's not .NET. WinRT is a language-independent set of deeply integrated OS components that use .NET metadata to make their APIs available to a wide variety of programming languages. Figure 3.3 shows the relationship between the languages and the metadata.

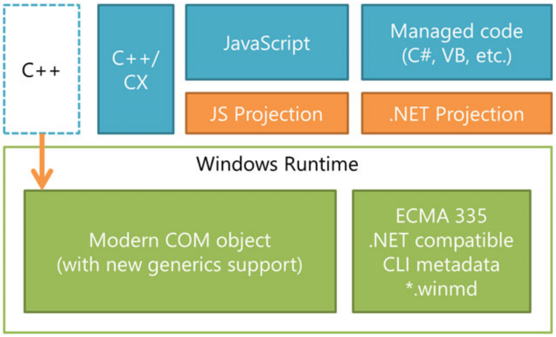

C++
C++/CX
JavaScript
Managed code (C#, VB, etc.)
JS Projection
.NET Projection
Windows Runtime
Modern COM object (with new generics support)
ECMA 335 .NET compatible CLI metadata *.winmd

Figure 3.3 JavaScript and managed code both use their language projections, which are based on the .NET-compatible metadata for a given component or set of components. You can even view the contents of a .winmd file using ILDASM. You can also code against the WinRT COM objects using C++ and the Windows Runtime Template library, but the recommended approach is to use C++ with the Component Extensions (C++/CX).

WinRT is to Windows Store apps as Win32 is to the desktop. Each is considered to be the lowest level practical API for its platform. Much like Win32, WinRT calls typically resolve down to native kernel calls, or for visualization, to DirectX.

But that's not the full picture. Unlike Win32, WinRT was designed from the start to be easily consumed by managed, native, scripting, and other types of languages all while meeting a high performance bar. WinRT takes the best of COM[1] (Component Object Model) and adds three important concepts:

- *Metadata*—WinRT components include a version of Common Language Infrastructure (CLI)[2] metadata. This metadata enables WinRT components to be self-describing.

---

[1] I know, as soon as I said "the best of COM," a bunch of you burst out laughing. COM is not anywhere nearly as bad an API model as many make it out to be. Most of its bad reputation comes from DCOM or from the dependency on the registry for system-wide COM objects.

[2] http://en.wikipedia.org/wiki/Common_Language_Infrastructure and http://en.wikipedia.org/wiki/.NET_metadata

- *Projections*—Each language is responsible for having projections that make WinRT work for the specific language.
- *Multiple languages*—The reason why metadata and projections were included was to easily enable support for multiple languages. More on that in a moment.

In this section, I'll cover COM and then the basics of WinRT metadata, how projections work, and how the different languages plug into the platform.

3.2.1 COM: back to the future

WinRT is native code. Not only that, it's the newest flavor of native COM code. COM is the interop model that started as OLE (Object Linking and Embedding) in early versions of Windows and eventually expanded to become DCOM (Distributed COM) and COM+ (which itself started life conceptually in Microsoft Transaction Server). COM is what enables you to put an ActiveX control on a web page or to automate Microsoft Word from your desktop application. COM is also the force behind those fun HRESULT error messages that crop up from time to time in various applications on your PC.

COM is a Windows technology. People have tried to implement it on different platforms, but fundamentally it relies on Windows infrastructure, including interprocess communication, the registry, and much more.

Unlike the C-style flat APIs in Win32, COM is object/type and interface based. You can implement flat APIs in COM, but you can also create object-oriented APIs that better encapsulate the functionality you wish to offer.

I tell you all this to point out that COM itself is not a new technology. It's old and stable. It's proven, and when used without brokers or late-binding, it's very fast compared to managed code. C++ developers use it natively all the time, because many of the newer Windows APIs are surfaced through COM.

WinRT components are essentially COM components without an `IDispatch` interface. A friend (who wished not to be identified) showed a room full of people how to create WinRT components using straight C++ COM APIs, skipping metadata, and skipping anything else that might otherwise make it look obviously WinRT-ish. Rather than using `CoCreateInstance`, (COM Object) functions like `RoActivateInstance` (Runtime Object) were used.

AN EVOLUTION

That's not to say that COM hasn't evolved in all this time. When it was first under development, the team called it by a number of names, one of which was Modern COM, or MoCo. To support WinRT, COM added long-awaited support for .NET-style generics. As you'll see in the next section, it's not the only .NET feature the team decided was worth using. Within Microsoft, this is a big deal, because COM and .NET were two competing technologies—WinRT really helps bring them together.

Another capability that was added comes in the form of the `IInspectable` interface. It's this interface that provides the metadata to the code implementing the projection, primarily for JavaScript and other dynamic languages. C++ components must

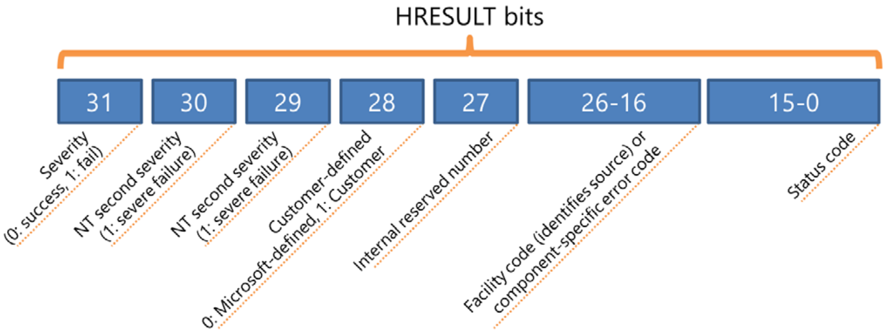

HRESULT bits
31
30
29
28
27
26-16
15-0
Severity (0: success, 1: fail)
NT second severity (1: severe failure)
NT second severity (1: severe failure)
Customer-defined 0: Microsoft-defined, 1: Customer
Internal reserved number
Facility code (identifies source) or component-specific error code
Status code

Figure 3.4 HRESULTs are back! It's useful to know what they actually mean.

implement this interface in order to be considered compatible with WinRT and be available to other languages.

There's a lot more there interface-wise, but this book isn't about COM development or WinRT apps in C++, so I'll skip the heavy details. There's more information in MSDN should you desire to really know what's going on under the covers from a C++ perspective.

ERROR REPORTING

One thing COM doesn't have, unfortunately, is a good .NET-style exception infrastructure. COM components still return HRESULTs (error handles), which often don't provide much in the way of help when debugging. Windows itself has communication mechanisms built in to provide error context to debuggers (not runtime). In addition, the WinRT and .NET teams have done a lot of work to help minimize our exposure to raw HRESULTs, but we'll still see them from time to time, especially in this first version of WinRT. Knowing that, it's useful to understand what an HRESULT actually is, although most of us will simply Google/Bing the full number. Figure 3.4 shows the breakdown of this 32-bit number.

PERFORMANCE IMPROVEMENTS

WinRT implements a number of performance improvements that weren't there in previous versions of COM. For example, many of the APIs are now asynchronous. This helps us maintain a fluid UI without the jitter and lag associated with tying up the UI thread with long-running operations.

Another improvement is to the infrastructure itself. WinRT has a new HSTRING type to help manage string transfers across boundaries and mapping them to C++, .NET, and JavaScript native string types. In COM in the past, passing strings across the managed/native boundary was a source of performance issues. In fact, this new string type is called a fast-pass string. An interesting implication of this is that the specific performance characteristics of `System.String` in .NET that you've learned over the years may simply no longer apply when communicating with WinRT components. Any string that's going to be passed to a WinRT API is allocated in Windows, as opposed to in the .NET Framework. The compiler handles this for you transparently.

One of the biggest additions to COM is the self-describing capability provided by the infrastructure and by an unexpected ally: .NET metadata.

3.2.2 Metadata

Standard Win32-style flat DLLs aren't self-describing. You have to know their entry points, that is, the functions they define. The result of this is a proliferation of constructs used to access the functions and entire websites dedicated to defining the API signatures.[3] Seemingly half of MSDN is dedicated to just descriptions of the parameters and types used to invoke native Win32 calls.

That's hardly programmer friendly and, quite honestly, was a problem that was solved with type libraries and with .NET in general. IntelliSense and the object browser are my primary sources of documentation. What makes that possible? If you guessed "metadata," then you're correct. Metadata is so cool, in fact, that I have a shirt with it right on the front (figure 3.5).

Figure 3.5 Metadata is cool. Yes, that is my shirt. Yes, my wife lets me dress like that in public. \m/

CLI metadata enables a component to state all the types it makes available to outside components. That includes all the classes, structs, functions, methods, enums, and so on that it includes. This metadata is what .NET reflection works from when providing information about a .NET assembly.

But unlike a .NET assembly, a native code component can't contain usable embedded metadata. Because all of the built-in WinRT components are native COM components, the metadata needs to be external in a .winmd file. This is similar in concept to the external type libraries (.tlb) we used to run into so often when consuming components in VB6 and the early days of .NET.

One interesting aspect of the metadata is that because it's .NET compatible, you can use existing tools such as ILDASM (.NET Intermediate Language Disassembly tool) to inspect it. To try this out yourself, first pull up a Visual Studio 2012 command prompt. If you don't already have this pinned to your start screen or taskbar, simply type `command` from the Start screen and pick one of the four VS2012-specific entries that come up. Figure 3.6 shows how.

On my machine, ILDASM is located in C:\Program Files (x86)\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools\ildasm.exe. Of course, you can navigate directly to it and run it from there, but knowing how to get to the VS2012 command prompt is useful. Once ILDASM is open, select File > Open and browse to the Windows.winmd file. On my Windows 8 64-bit machine, it's located here: C:\Program Files (x86)\Windows Kits\8.0\References\CommonConfiguration\Neutral.

---

[3] "A wiki for .NET developers," Do interop the wiki way, http://pinvoke.net.

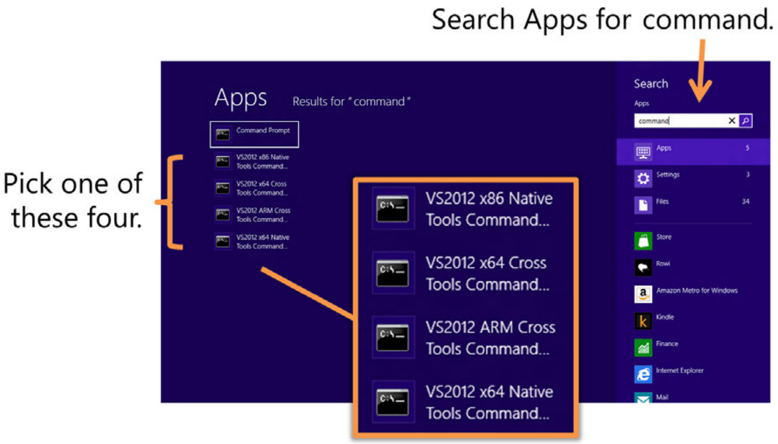
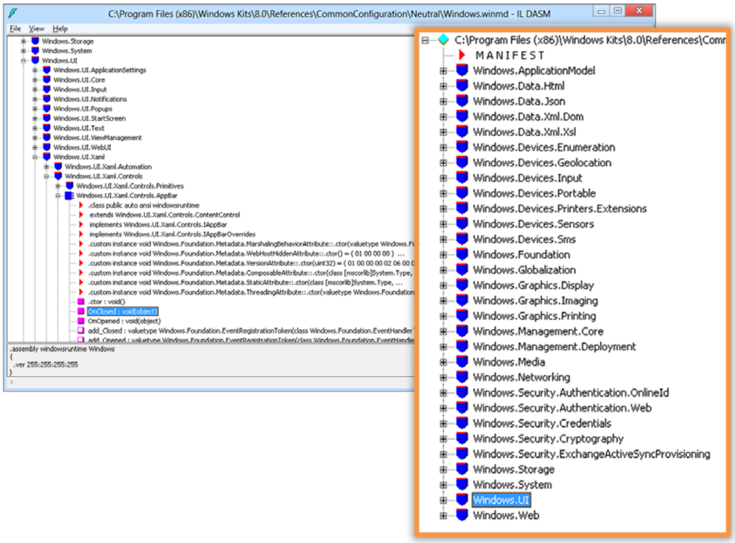

Search Apps for command.
Pick one of these four.
Apps
Results for "command"
Command Prompt
VS2012 x86 Native Tools Command...
VS2012 x64 Cross Tools Command...
VS2012 ARM Cross Tools Command...
VS2012 x64 Native Tools Command...
Search
Apps
command
Apps 5
Settings 3
Files 34
Store
Rowi
Amazon Metro for Windows
Kindle
Finance
Internet Explorer
Mail

Figure 3.6 On the Start screen, type command to pull up a list of command prompts. On a full Visual Studio 2012 Premium or Ultimate install, you'll see four options plus the generic command prompt. Pick any of the four to get a command prompt with the VS2012 tool locations added to the path.

The Windows.winmd file contains the metadata for the entire WinRT API—everything in the Windows.* namespaces when you look in the object browser. Figure 3.7 shows ILDASM with the WinRT metadata library loaded.

Despite using CLI metadata, WinRT isn't a managed platform. You previously learned that it's built using native code. As much as I like .NET, imposing the overhead of a JIT compiler, runtime startup, and periodic garbage collection to platforms such as C++ and DirectX wouldn't make anyone happy. Instead, WinRT, being built on COM, is a reference-counted platform. That is, when you obtain a reference to a WinRT


C:\Program Files (x86)\Windows Kits\8.0\References\CommonConfiguration\Neutral\Windows.winmd - IL DASM
File
View
Help
Windows.Storage
Windows.System
Windows.UI
Windows.UI.ApplicationSettings
Windows.UI.Core
Windows.UI.Input
Windows.UI.Notifications
Windows.UI.Popups
Windows.UI.StartScreen
Windows.UI.Text
Windows.UI.ViewManagement
Windows.UI.WebUI
Windows.UI.Xaml
Windows.UI.Xaml.Automation
Windows.UI.Xaml.Controls
Windows.UI.Xaml.Controls.Primitives
Windows.UI.Xaml.Controls.AppBar
.class public auto ansi windowsruntime
extends Windows.UI.Xaml.Controls.ContentControl
implements Windows.UI.Xaml.Controls.IAppBar
implements Windows.UI.Xaml.Controls.IAppBarOverrides
.custom instance void Windows.Foundation.Metadata.MarshalingBehaviorAttribute::.ctor(valuetype Windows.Fo
.custom instance void Windows.Foundation.Metadata.WebHostHiddenAttribute::.ctor() = (01 00 00 00) ...
.custom instance void Windows.Foundation.Metadata.VersionAttribute::.ctor(uint32) = (01 00 00 00 02 06 00 0
.custom instance void Windows.Foundation.Metadata.ComposableAttribute::.ctor(class [mscorlib]System.Type,
.custom instance void Windows.Foundation.Metadata.StaticAttribute::.ctor(class [mscorlib]System.Type, ...
.custom instance void Windows.Foundation.Metadata.ThreadingAttribute::.ctor(valuetype Windows.Foundation
.ctor : void()
OnClosed : void(object)
OnOpened : void(object)
add_Closed : valuetype Windows.Foundation.EventRegistrationToken(class Windows.Foundation.EventHandler
C:\Program Files (x86)\Windows Kits\8.0\References\Comm
MANIFEST
Windows.ApplicationModel
Windows.Data.Html
Windows.Data.Json
Windows.Data.Xml.Dom
Windows.Data.Xml.Xsl
Windows.Devices.Enumeration
Windows.Devices.Geolocation
Windows.Devices.Input
Windows.Devices.Portable
Windows.Devices.Printers.Extensions
Windows.Devices.Sensors
Windows.Devices.Sms
Windows.Foundation
Windows.Globalization
Windows.Graphics.Display
Windows.Graphics.Imaging
Windows.Graphics.Printing
Windows.Management.Core
Windows.Management.Deployment
Windows.Media
Windows.Networking
Windows.Security.Authentication.OnlineId
Windows.Security.Authentication.Web
Windows.Security.Credentials
Windows.Security.Cryptography
Windows.Security.ExchangeActiveSyncProvisioning
Windows.Storage
Windows.System
Windows.UI
Windows.Web

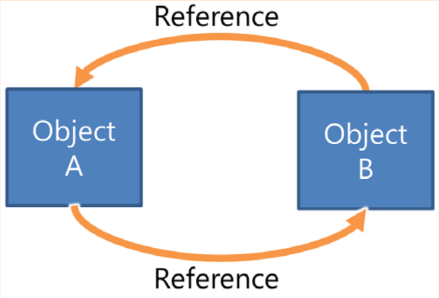
```
.assembly windowsruntime Windows
{
  .ver 255:255:255:255
}
```

Figure 3.7 ILDASM with the Windows.winmd file loaded and the Windows.UI node expanded to view the XAML definitions. From here, it looks like any other .NET assembly, but this is just metadata.

object, an internal counter is incremented. When you release the reference, the counter is decremented. When the counter reaches zero, the object cleans itself up.

Metadata alone doesn't seem sufficient to explain how these objects look so much like .NET. Something else must be going on here. What is it, you ask? The projection.

Potential for circular references



One of the goals of .NET was to eliminate the circular reference issue that can show up in reference-counted systems when two objects have references to each other, but nothing else references them, as illustrated at right.

In this example, Object A has a reference to Object B. Therefore, Object B's reference count is at least 1. Object B then has a reference to Object A; therefore Object A's reference count is at least 1. These objects would never automatically clean themselves up until at least one of the references is manually removed.

This is something COM developers have dealt with for ages in return for not having the overhead of garbage collection. I point this out just to remind you that although the objects look like .NET objects, and because of the projections, even seem to behave like .NET objects, they're not .NET objects.

WinRT components are careful about avoiding circular references among themselves, but if you create your own WinRT extension components (in chapter 21, when I show how to integrate with Xbox controllers), you'll need to pay attention to the possibility of circular references.

3.2.3 Projections

Projections build on metadata to provide language-specific versions of the WinRT APIs. They're purpose-built by the language implementation teams to provide a high-performance version of the API.

In the .NET world, projections are thin wrappers, much like the Runtime Callable Wrappers (RCWs) we have when importing COM components into .NET. With the exception of string processing and some other performance enhancements, they're almost identical to classic RCWs.

One of the more interesting things about projections is that they take on the characteristics of their target languages' standards. For example, the JavaScript projections use JavaScript `camelCase` form for methods. The same APIs in the .NET projections use the familiar `PascalCase` form.

There's even more to it than that (especially around asynchronous function handling, which we'll discuss in chapter 16), but let it suffice to say that the projections, which are maintained by Windows teams and which are updated in lock-step with the platform, are your interface to WinRT from the managed and scripting languages supported by the platform.

I've mentioned several times that WinRT was created to be consumed efficiently from many different types of languages, including native, scripting, and managed. In the next section, we'll look at the current choices we have when deciding which to pick.

3.3 Client technologies and languages

WinRT was designed from the start with multiple language support in mind, more so than even straight COM was. In the previous section, you found out that the language projection is the way that WinRT is exposed to the other languages. Therefore, the list of languages supported by WinRT is defined by the availability of an appropriate language projection.

Out of the box, Microsoft provides the languages and presentation layers shown in table 3.1.

Table 3.1 Languages and presentation technologies that can create Windows Store Modern Style apps

Language	Presentation	Description
C#/VB	XAML	C# and VB languages using the Windows Runtime and .NET 4.5, with the presentation layer created in XAML. The resulting application code looks very similar to Silverlight. This is the approach most current .NET and Silverlight/WPF developers will gravitate to, and it's the one I'll cover in this book.
C++/CX	XAML	C++/CX using the Windows Runtime and XAML. This approach enables you to use existing C++ code or skills combined with the power of XAML for creating the interface. It's possible to combine this approach with, DirectX for the best of both worlds. Most code is standards-based C++, with only the WinRT interface code (such as that using XAML) using CX.
C++/CX	DirectX	Primarily used for games or performance-sensitive visualizations, this approach uses C++ along with the Windows Runtime and DirectX libraries for frame-based graphics applications. If you've typically built XNA applications in the past, this is the route to take in Windows. As with XAML, most code is standards-based C++, with only the WinRT interface code using CX.
JavaScript	HTML/CSS	A new and exciting way to create native client-side Windows applications using HTML and CSS for the presentation layer, along with the Windows Runtime and JavaScript for the code.

The parity between the capabilities, with the exception of DirectX, is impressive. You can't do more with C#/XAML than you can with a JavaScript/HTML, or vice versa. They're equally powerful in terms of system access. A notable standout is C++ and DirectX. The only way to use DirectX in Windows Store apps is to use C++ (or a C++ wrapper around DirectX); XNA isn't supported for Windows Store apps. Other than DirectX, C++ is still required to use the same whitelisted Win32 calls and Windows Runtime API that C#, VB, and JavaScript applications use.

Both C++ and C#/VB can create WinRT extension assemblies that may be used by any other supporting language. For performance reasons, when creating an extension assembly, you'll probably want to do it in C++; otherwise you incur the overhead of the CLR spinning up. It's through extension assemblies that you can use features available only in C++ (such as DirectX) from a .NET C# application.

Which technology you choose then comes down to which language and presentation you're most comfortable with. You no longer need to choose capability over familiarity. Regardless of your comfort zone and preferences, as a developer you have a solid set of equally capable languages and presentation styles from which to choose.

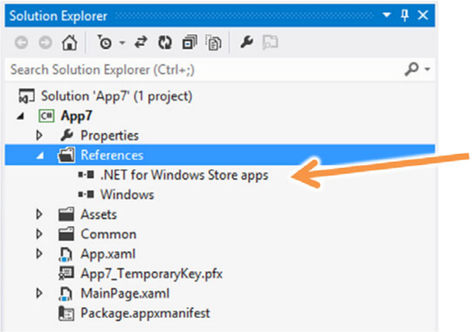
Of course, I'm going to make the assumption that readers of this book have chosen C# (or VB) .NET and XAML. With that understood, let's take a look at some of the interesting WinRT and .NET 4.5 APIs available to us.

3.4 A brief tour of WinRT and .NET 4.5

When you create a Windows Store app project, the Windows Store app profile for .NET 4.5 and the entirety of WinRT are already referenced and available to you, as shown in figure 3.8. They're guaranteed to be on the machine you deploy to, and because you don't have to package them with your app, auto-referencing them carries no real cost.

The .NET 4.5 subset made available to Modern Style apps is commonly referred to simply as .NET for Windows Store apps. It's not a reimplementation of .NET as Silverlight was but rather a set of shadow assemblies that point right back into the main .NET 4.5 installation on the machine. That means you're using real .NET, with the only limitations made being for security, execution speed/smoothness, and appropriateness (and, of course, eliminating things that duplicate WinRT functionality), not for download size or cross-platform capability.

I'm not going to duplicate all of MSDN here, but I did want to call out a few interesting namespaces in both .NET 4.5 and WinRT. The rest of this book will show how to use a large number of APIs in WinRT and in .NET 4.5 as well, but some deserve calling out in advance. Table 3.2 shows some that I find good to know from the start. Remember, anything that starts with `Windows.*` is WinRT, everything else is .NET.



The references are already there.

Figure 3.8 When you create a project, the entirety of the platform is already referenced for you. Because these are guaranteed to exist on the target machine, there's no deployment penalty.

Table 3.2 A few interesting WinRT and .NET 4.5 APIs

Namespace	Why it's interesting
`Windows.UI.Xaml.*`	This is where you'll find almost the entire XAML stack. Yes, it's all in native code, and it's built into the core API of the platform.
`Windows.Data.Json`	Includes basic JSON parsing functionality. I've found that most people prefer to use another library like JSON.NET instead.
`Windows.Data.Xml.*`	The XML DOM parser and XSLT processor are both found here.
`Windows.Devices.*`	All the device access functionality is in here.
`Windows.Foundation`	You'll see more of this in the next chapter. This is where the asynchronous support code lives.
`Windows.Networking.*`	This is where much of the support for networking lives. For most typical networking work you do, however, you'll use the .NET classes. If you're looking for sockets, websockets, background transfers, push notifications, or basic connectivity information, you'll find it here.
`System.Net.*`	`HttpClient`, `Cookie`, `HttpWebRequest`, `WebRequest`, and all the usual .NET networking classes are located here, with the exception of sockets, which is in `Windows.Networking`.
`Windows.Storage.*`	`Windows.Storage` has much of what you used to rely on `System.IO` for in .NET.
`Windows.Web.Syndication`	RSS processing. You can find `AtomPub` processing in `Windows.Web.AtomPub`.
`Windows.Security.*`	If you need to authenticate a user, identify a user, or work with cryptography, this is the root namespace you'll need.
`Windows.Media.*`	Many apps need to work with video or audio. As you'd expect, this is built right into WinRT and made available to any app.
`System.Linq.*`	`Linq` remains in .NET, not in WinRT.
`System.Runtime.Serialization.Json`	JSON serialization/deserialization.
`System.Runtime.Serialization.Xml`	XML serialization/deserialization.
`System.Runtime.WindowsRuntime`	To make working with WinRT easier for .NET developers, it made sense to include a number of extension methods that work on WinRT types. You'll find them here. Note that the namespaces inside this assembly differ from the assembly name.

Table 3.2 A few interesting WinRT and .NET 4.5 APIs *(continued)*

Namespace	Why it's interesting
`System.Runtime.WindowsRuntime.UI.Xaml`	This includes a small amount of the XAML stack that's implemented in .NET rather than native code in WinRT. These are generally structures and enumerations, not rendering, controls, or other core XAML functionality.
`System.Threading.Tasks.*`	This is the Task Parallel Library. It's not only built into .NET 4.5, but it's also a core part of how you interact with asynchronous code in WinRT.

One thing you may immediately notice is just how much of .NET has been refactored into WinRT and native code. It's fast, it's available to more than just .NET apps, and it's guaranteed to be there. There's much more than I included in this table, of course—even the .NET subset available to Windows Store apps is huge. We'll touch on a good bit of it throughout the rest of this book.

3.5 Summary

Windows 8 offers a brand-new platform for application development. For those of us who like to code in .NET and design in XAML, that platform revolves primarily around .NET and the Windows Runtime. WinRT is the newest flavor of COM with the addition of .NET metadata. Metadata makes WinRT discoverable and usable by a number of languages, including JavaScript, C#/VB, and C++.

Individual languages use projections to make the COM-oriented WinRT API friendly to them. In the case of .NET, these projections are essentially Runtime Callable Wrappers, much like we've used to access COM components in the past. The WinRT API takes over a large amount of what used to be .NET code in `System.*` and moves it to native code in `Windows.*`.

The developer improvements in Windows 8 are much larger than just a new API, however. Windows 8 includes a new application deployment model, the .appx, and an app store that knows how to sell them. The Windows app store verifies that the application in the .appx is safe to use (and that it adheres to the sandbox) and makes it available to consumers to use. Unlike desktop development, you can't simply send someone a zip file with your compiled application; you need to use known and trusted deployment mechanisms.

In addition to the deployment model, Windows 8 includes a new driver model. The new model helps keep the environment safe and performing well by restricting access to devices that haven't been specifically produced with WinRT and Windows Modern Style apps in mind.

In the next chapter, we'll look at one of the most important namespaces in WinRT: `Windows.UI.XAML`.

4

XAML

This chapter covers

- XAML elements and namespaces
- Properties, events, and commands
- Object trees and namescope

XAML (Extensible Application Markup Language) is a declarative language that enables you to create and initialize objects using XML. This approach to development was popularized by the WPF and Silverlight, as well as by Silverlight on Windows Phone.

Everything you can do in XAML, you can do in code. But to make the most of the platform and its tooling, you'll want to embrace the code-plus-markup philosophy rather than go with a 100% code solution. The road to this approach has been well trodden by Silverlight, WPF, and Windows Phone, so I'm completely comfortable in saying code plus markup should be the default approach for almost every application you write.

> **NOTE** The markup language, the property system, and the things that make those two work together are collectively referred to as WinRT XAML, mostly because it's easier than saying "C# or VB with .NET and the Windows Runtime using XAML for presentation." Trust me on that.

With Windows 8, for the first time XAML is built in at the OS level. Every Windows 8 machine will have WinRT XAML, a native implementation optimized for the Windows platform. This is in contrast to WPF, which required a download of the .NET Framework for a mostly managed implementation of XAML, and Silverlight, which had similar requirements. It's nice to know that Microsoft both believes in the appropriateness of XAML and is willing to put in the extra effort to make sure it has an excellent implementation in Windows.

This chapter will cover all the XAML basics you need to know to make use of the rest of the building blocks in this book. I'll start with those things that give XAML its structure: elements and namespaces.

From there, a dive into the property system is required in order to understand dependency and attached properties and how they enable binding and animation. The types of properties you'll use in XAML aren't the same as you may be using; they have a different underlying implementation. Along the way, we'll also look at a few different ways to specify properties in markup.

I'll wrap up this chapter with a look into the object trees that result from parsing XAML and the namescopes functionality required to prevent naming collisions.

First, to understand the structure of a XAML file, it's important to understand the representation and use of objects, namespaces, properties, and events.

4.1 Elements and namespaces

Any XML file is made up of individual elements each enclosed in angle brackets. By default, there are no known elements; for schema-validated XML, they all must be defined in a schema somewhere, identified by a namespace. XAML, being implemented in XML, follows these rules.

In this section, I'll first present the implementation of elements as used in XAML. You'll see how elements have a 1:1 correspondence with objects in the runtime. You'll also learn how they are instantiated and how they're nested.

After the discussion of elements, I'll cover namespaces. Many people who have used XML have never run across XML namespaces, because they didn't work against a schema. In XAML, anything you import must exist in a namespace, so understanding how to use them is fundamental to making XAML work.

I'll start with looking at how elements and code relate.

4.1.1 Objects as elements

The XAML format enables you to easily visualize a hierarchy of elements while separating presentation from code. XAML represents types and properties, not logic. In XAML, each element maps to a .NET or WinRT type. Similarly, each attribute within an element corresponds to a property of that type. For example, these statements are functionally equivalent:

```
<TextBlock x:Name="tb" Text="Hello World!" />
```

```
TextBlock tb = new TextBlock();
tb.Text = "Hello World!";
```

The `TextBlock` in XAML is a representation of the same class as the `TextBlock` in code. Note also that the `TextBlock` element in the XAML implicitly calls the constructor that you explicitly call in code. This initialization occurs because, each time an element is created in XAML, the corresponding type's default constructor is called behind the scenes.

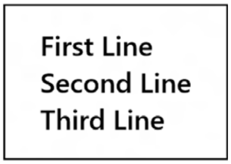
> **TIP** In XML, and therefore in XAML, elements are surrounded by angle brackets. Attributes are surrounded by quotes whether they are strings, numbers, dates, or anything else.

Objects (or instances of types) are represented in XAML using XML elements. The elements have the same name as the associated class and are instantiated by the XAML parser at runtime.

> **NOTE** Any type you use in XAML must have a default (parameterless) constructor. Windows Runtime XAML currently has no provision for passing arguments into a constructor or an initialization function, so you'll need to make sure your types can be initialized using defaults and properties alone.

Certain types of objects may contain one or more of other nested objects. For example, a button may contain a single content object, which itself may contain one or more other objects. Figure 4.1 shows three lines of text in a button.

In the following listing, the result of which is shown in figure 4.1, the `Page` contains the `Grid`, the `Grid` contains the `Button`, and the `Button` contains a `StackPanel`, which is a panel that by default lays its children out in a vertical list. The `StackPanel` itself contains three `TextBlock` elements. (`Grid` and `StackPanel` layout will be covered in detail in the next chapter.)



First Line
Second Line
Third Line

Figure 4.1 The button from listing 4.1 showing three `TextBlocks` nested inside a `StackPanel`

Listing 4.1 XAML showing a hierarchy of nested objects

```
<Page
  x:Class="XamlExample.MainPage"  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:XamlExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Button Height="100" Width="150" Margin="10">
      <StackPanel>
        <TextBlock Text="First Line" />
        <TextBlock Text="Second Line" />
        <TextBlock Text="Third Line" />
      </StackPanel>
    </Button>
  </Grid>
</Page>
```

Three TextBlocks in StackPanel

The `Panel` and `Button` are both content controls, which can contain a composition of other elements. It's important to understand that a content control can only have one direct child element, typically a panel (`StackPanel`, `Grid`, `Canvas`, and so on) that holds the other elements. In this case, the `Button` contains a `StackPanel`, which then includes the real content.

The `Grid` and `StackPanel` are both `Panel`s, which is a type that has a `Children` collection to allow multiple contained elements. The `x:Name` and `x:Class` properties are part of the namespace specified by the `xmlns:x` statement, which I'll cover in the next section. We'll discuss panels in detail in chapter 6.

> **NOTE** In many of the examples in this book, I use black text on a white background, all hardcoded in the XAML. This is a concession to print, where a black background makes things difficult to read. In your own work, you'll usually want to go with the color settings as provided in the templates. By default, that's white text on a dark charcoal gray background. WinRT provides standard dark/light theme colors but currently doesn't provide access to the user's color preferences as set in the control panel.

The ability to flexibly nest objects permits a composition approach to UI design. Rather than having to purchase or custom-code a button control that allows, say, three lines of text and an image, you can simply compose those into an appropriate layout panel and make that panel the content of the button control.

This nesting of objects is part of what gives you an object tree. We'll cover that in more detail later in this chapter.

Now that we've covered the basic structure of a XAML file, let's talk about how you differentiate your `AwesomeButton` control from my `AwesomeButton` control, even though we used the same control name: namespaces.

4.1.2 Namespaces

Namespaces in XAML, which are just XML namespaces, are similar to namespaces in other languages such as C# and Java. To specify where to look for an element, you reference a namespace in XAML either on that element or more typically at the root element of the XAML file. Namespaces can be used on the element they're declared on as well as in any nested element. The following listing illustrates the use of the several different namespaces, all declared at the root `Page`.

Listing 4.2 Namespaces in XAML

```
<Page
  x:Class="NamespaceExample.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:NamespaceExample"
  >

  <Grid>
    <local:RadialLayoutPanel>
```

Standard XAML namespace

Declare namespaces

Use local namespace

```
      <TextBlock Text="First Item" />
      <TextBlock Text="Second Item" />
      <TextBlock Text="Third Item" />
      <TextBox x:Name="UsernameField" />
    </local:RadialLayoutPanel>

  </Grid>
</Page>
```

Use x namespace

Every XAML file has the default namespace declared using `xmlns` without any prefix character. Most XAML files also have other namespaces such as `d` and `mc`, not shown here. Those are used to provide design-time specific behavior in Visual Studio and Expression Blend.

The `x` namespace is used to provide the `Name` property for elements as well as the `Key` property for items in a dictionary. There's nothing special about the use of `x`, that's just the standard convention; you could name it `foo` if you wanted, although I don't recommend it. Technically you can even use just `Name` on elements, but for the best compatibility with Silverlight and WPF XAML, you'll want to use `x:Name` instead.

In an application of any real complexity, there'll come a time when you need to reference classes declared either in your own project or in an external referenced library. This is accomplished using the `using` statement followed by the code namespace the class is declared in. In this example, the local namespace refers to the root of this project. If I had another namespace named `Controls` in the project, I could declare the namespace like this:

```
xmlns:localControls="using:NamespaceExample.Controls"
```

You can, of course, call the prefix anything you want. In keeping with past practices, I named it `localControls` to help differentiate it from controls imported from other assemblies. There are certain common conventions, such as the use of `local` and `x`, but otherwise, how you name your prefixes is up to you.

> **TIP** The `using` statement is new to WinRT XAML. If you're coming from Silverlight or WPF, you may be used to using the `clr-namespace` approach. This approach is no longer appropriate in WinRT XAML because elements can come from Windows Runtime libraries, not just from .NET assemblies.

Except for the default namespace, which lacks a prefix, all namespaces declared in XAML are used by using the `<prefix>:<element or attribute>` syntax, several examples of which may be seen in listing 4.2.

Elements and namespaces make up most of the structure of XAML. Elements provide a way to instantiate objects in the runtime or in your own code. Namespaces provide a way to import objects from different libraries. Both are essential elements of XAML.

You can accomplish a lot with just elements grouped by namespaces. But you'll typically need a way to work with the individual properties of the objects/elements.

4.2 Properties

Most classes we develop expose properties that can be manipulated by consuming code. For example, a `Person` class might expose a `LastName` property. Similarly, the `TextBlock` class exposes, among many others, a `Text` property. These properties provide a wrapped way to manipulate the data associated with a class.

In WinRT XAML, properties are even more important because they can be the source or target of data binding, or the target of an animation, or more. For those reasons, a special property system had to be developed.

In this section I'll cover how to use properties in XAML. First, I'll explain the different ways of specifying properties using both property element syntax and XML attributes. From there, I'll go into the dependency property system and how it works with the other WinRT XAML subsystems. At the same time, we'll look at a specialized type of dependency property called an attached property.

Finally, we'll take a brief look at property paths—something that will be essential when you look at control templates or animation.

Let's start by figuring out how to specify property values in markup.

4.2.1 Property syntax

There are two ways to reference properties in XAML: inline with the element as you would any XML attribute, or as a nested subelement. Which you should choose depends on what you need to represent. Simple values are typically represented with inline properties, whereas complex values are typically represented with element properties.

The use of an inline property requires a type converter that will convert the string representation—for example, the `"Black"` in `Background="Black"`—into a correct underlying type (in this case, a `SolidColorBrush`). The example in the following listing shows a built-in type converter being used to convert the string `"Black"` for the inline property `Background`. The type converter is built into the system and invoked automatically.

Listing 4.3 Specifying a property value inline using an XML attribute

```
<Page
  x:Class="PropertyExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PropertyExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid Background="Black"
        Width="600">

  </Grid>
</Page>
```

Inline properties

This listing also shows a numeric property, `Width`, which gets converted to a double from the string `"600"`. Built-in converters can be extremely simple as in the numeric case or more complex as in the case of the brush converter or, as you'll see in later chapters, the mini-language that specifies points in geometry.

Another way to specify properties is to use the expanded property element syntax. Although this can generally be used for any property, it's typically required only when you need to specify something more complex than the inline syntax will easily allow. The syntax for element properties is `<Type.PropertyName>value</Type.PropertyName>`, as shown here.

Listing 4.4 Specifying a property value using property element syntax

```
<Page
  x:Class="PropertyExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PropertyExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid Width="600">
    <Grid.Background>
      Black
    </Grid.Background>
  </Grid>
</Page>
```

Element property

The use of the string to invoke the type converter is, in its end result, identical to using `<SolidColorBrush Color="Black" />` in place of `"Black"`. Though these examples are rarely seen in practice, the more complex example of setting the `Background` property to a `LinearGradientBrush` is common, so we'll cover that next.

Rather than have the brush represented as a simple string such as `"Black"` as shown in the previous listing, the value can be an element containing a complex set of elements and properties such as the `<LinearGradientBrush>` shown in the next listing.

Listing 4.5 A more complex example of the property element syntax

```
<Page
  x:Class="PropertyExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PropertyExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
  <Grid Width="600">
    <Grid.Background>
      <LinearGradientBrush>
```

Background property

```
            <LinearGradientBrush.GradientStops>
              <GradientStop Offset="0.0" Color="Black" />
              <GradientStop Offset="0.5" Color="Blue" />
              <GradientStop Offset="0.5" Color="Orange" />
              <GradientStop Offset="1.0" Color="DarkGray" />
            </LinearGradientBrush.GradientStops>
          </LinearGradientBrush>
        </Grid.Background>
      </Grid>
    </Page>
```

In this example, you expand the `Background` property using property element syntax. You then have the ability to nest a complex type, such as the `LinearGradientBrush`, within it.

Now that you know how to specify properties in markup, let's dive deeper into how those properties work.

4.2.2 Dependency properties

Dependency properties are part of the property system introduced with WPF and Silverlight and used in WinRT XAML. In markup and in consuming code, they're indistinguishable from .NET properties, except that they can be data bound, serve as the target of an animation, or be set by a style.

> **TIP** A property can't be the target of an animation or obtain its value through binding unless it's a dependency property. We'll cover binding in detail in chapter 8.

To have dependency properties in a class, the class must derive from `DependencyObject` or one of its subclasses. Typically, you'll do this only for visuals and other elements that you'll use within XAML and not in classes defined outside of the user interface.

In regular code, when you create a property, you typically back it by a private field in the containing class. Storing a dependency property differs in that the location of its backing value depends on its current state, and the CLR property wrapper is just a convenience, as shown in figure 4.2. The way that location is determined is called *value precedence.*

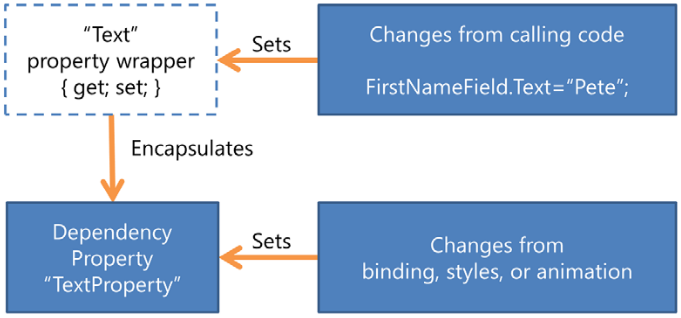

"Text"
property wrapper
{ get; set; }
Sets
Changes from calling code
FirstNameField.Text="Pete";
Encapsulates
Dependency
Property
"TextProperty"
Sets
Changes from
binding, styles, or animation

Figure 4.2 Dependency properties aren't backed by private fields like normal CLR properties. Instead, the CLR property wrapper is just a convenience. Inside the wrapper, it uses the `SetValue` and `GetValue` methods to access the dependency property itself.

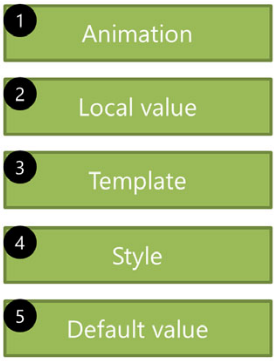


Figure 4.3 Value precedence for dependency properties. Value changes from animation take precedence over locally set values, which have precedence over template values, and so on. This precedence allows properties to have meaningful value changes in the app and also respond correctly to setting values in markup and code.

VALUE PRECEDENCE

Dependency properties obtain their value from a variety of inputs. What follows is the order the XAML property system uses when assigning the runtime values of dependency properties, with the highest precedence listed first, as shown in figure 4.3.

Here's the precedence in more detail:

- *Active or hold animations*—Animations will operate on the base value for the dependency property, determined by evaluating the precedence for other inputs. In order for an animation to have any effect, it must be highest in precedence. Animations may operate on a single dependency property from multiple levels of precedence (for example, an animation defined in the control template and an animation defined locally). The value typically results from the composite of all animations, depending on the type being animated. If you think about animating the position of an element, you'll want that animated value to take precedence over one set in code or markup. The property system helps ensure that it happens.
- *Local value*—Local values are specified directly in the markup and are accessed via the property wrappers for the dependency property. When you directly assign a value to a property in XAML or in code, that's a local value. Because local values are higher in precedence than styles and templates, they're capable of overriding values such as the font style or foreground color defined in the default style for a control.
- *Templated properties*—Used specifically for elements created within a control or data template, their value is taken from the template itself.
- *Style setters*—These are values set in a style in your application via resources defined in or merged into the `UserControl` or application resource dictionaries. We'll explore styles in chapter 9.
- *Default value*—This is the value provided or assigned when the dependency property was first created. If no default value was provided, normal runtime defaults typically apply.

There are other subtleties to this. For example, controls typically have a default implicit style, which itself sets property values and includes a control template. That control template may contain visual states, which are themselves implemented as animations in a bit of an Inception-esque nesting. If you understand the basics of value precedence, you can always figure out where a value is coming from. I still find it easier than trying to debug web pages with complicated CSS layouts and rules.

The strict precedence rules allow you to depend on behaviors within the runtime, such as being able to override elements of a style by setting them as local values from within the element itself. In the next listing, the foreground of the button will be `Red` as set in the local value and not `Black` as set in the style. The local value has a higher precedence than the applied style.

Listing 4.6 Dependency property precedence rules in practice

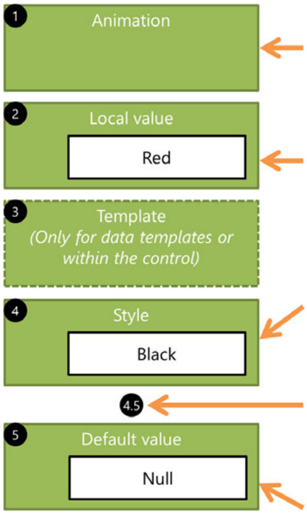
```
<Page
  x:Class="XamlExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:XamlExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Page.Resources>
    <Style x:Key="ButtonStyle"
           TargetType="Button">
      <Setter Property="Foreground"
              Value="Black" />
      <Setter Property="FontSize"
              Value="24" />
    </Style>
  </Page.Resources>
  <Grid>
    <Button Content="Local Values at Work"
            Style="{StaticResource ButtonStyle}"
            Foreground="Red" />
  </Grid>
</Page>
```

Style definition

Style in use

Local Value

The `Style` tag in `Page.Resources` is a reusable asset that sets some key properties for our button. Precedence in this case works very much like precedence in CSS: A locally declared style property overrides that defined at a higher level. The end result, in this case, is the precedence shown in figure 4.4.



There is no animation in our example.

The locally set value has precedence.

The locally defined style sets the property to "Black".

This is the point where the control's default style and its control template are considered.

The dependency property registration's default value.

Figure 4.4 The dependency property precedence for the `Foreground` property of the `TextBox` from listing 4.6. In this example, the `TextBox` ends up with a `Red` foreground.

For more information on dependency properties, please see the MSDN page here: http://bit.ly/WinRTXamlDP. You'll also implement your own dependency properties later in this book when you create custom controls and panels.

There's one other type of dependency property you must understand before you can truly grok the property system used in XAML. That type of property also has a slightly odd appearance and is called an *attached property*.

4.2.3 Attached properties

Attached properties are a specialized type of dependency property that's immediately recognizable in markup because of the `TypeName.AttachedPropertyName` syntax. For example, `Canvas.Left` is an attached property defined by the `Canvas` type. What makes attached properties interesting is that they're not defined by the type you use them with; instead, they're defined by another type in a potentially different class hierarchy. The class using the property doesn't need to have any knowledge of the class that defines the property.

Attached properties allow flexibility when defining classes because the classes don't need to take into account every possible scenario in which they'll be used and define properties for those scenarios. Layout is a great example of this. The flexibility of the layout system allows you to create new panels that may never have been implemented in other technologies—for example, a panel that lays out elements by degrees and levels in a circular or radial fashion versus something like the built-in `Canvas` that lays out elements by `Left` and `Top` positions.

Rather than have all elements define `Left`, `Top`, `Level`, and `Degrees` properties (as well as `GridRow` and `GridColumn` properties for grids) like you would in a technology like Windows Forms, you can use attached properties. The buttons in listing 4.7, for example, are contained in panels that have greatly differing layout algorithms, requiring different positioning information. In this case, we'll show a fictional `RadialPanel` in use (the panel doesn't exist, so the markup won't compile as is).

Listing 4.7 Attached properties in use

```
<Page
  x:Class="XamlExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:XamlExample"
  xmlns:panels="usingXamlExample.Panels"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <StackPanel>
    <Canvas Width="400" Height="200">
      <Button Canvas.Left="10"
              Canvas.Top="50"
              Width="200" Height="100"
```

Attached properties

```
              Content="Button in Canvas" />
    </Canvas>

    <panels:RadialPanel Width="400" Height="400">
      <Button panels:RadialPanel.Degrees="25"
              panels:RadialPanel.Level="3"
              Width="200" Height="100"
              Content="Button in Radial Panel" />
    </panels:RadialPanel>
  </StackPanel>

</Page>
```

Attached properties

Attached properties aren't limited to layout. You'll find them in the animation engine for things such as `Storyboard.TargetProperty` as well as in other places of the framework. For more information on attached properties, please see the MSDN page at http://bit.ly/WinRTXamlAttachedProps.

4.2.4 Property paths

Before we wrap up our discussion of properties, there's one concept left to understand: *property paths*. Property paths provide a way to reference properties of objects in XAML both when you have a name for an element and when you need to indirectly refer to an element by its position in the tree.

Property paths can take several forms and may dot-down into properties of an object. They can also use parentheticals for indirect property targeting as well as for specifying attached properties. Here are some examples of property paths for the `Storyboard` target property:

```
<DoubleAnimation Storyboard.TargetName="MyButton"
                 Storyboard.TargetProperty="(Canvas.Left)" ... />
<DoubleAnimation Storyboard.TargetName="MyButton"
                 Storyboard.TargetProperty="Width" ... />
...
<Button x:Name="MyButton"
        Canvas.Top="50" Canvas.Left="100" />
```

Path to an attached property: Canvas.Left

Path to a normal property: MyButton

Properties are one of the pieces that define an object's interface. When looking at XAML, most of what you see will be objects and their properties. But how does WinRT actually see all that? What's constructed from this information? XAML is simply a representation of objects and their properties, structured into an object tree.

4.3 Object trees and namescope

In the previous sections, I mentioned the concept of an object tree. In order to understand the object tree, you need to understand the layout and contents of XAML files. Once you do, it's easier to conceptualize the object tree and its related concept, namescope.

A common misconception is that the runtime creates XAML for any objects you create in code. In fact, the opposite is what happens: The runtime creates objects from XAML. Objects you create in code go right into the trees as their native object form. Elements in XAML are processed and turned into objects that go into the same tree.

4.3.1 Object trees

Now that we've covered the structure of a XAML file, you can look at one and quickly realize that it represents a hierarchical tree of objects starting from the root (typically a `Page`) and going all the way down to the various shapes, panels, and other elements that make up the control templates in use. That hierarchical structure is known as an *object tree*.

The following listing shows an example page's XAML. In this example, I have a grid with a couple of `TextBlock` elements and a `ListBox` element. For illustrative purposes, the `ListBox` has several `TextBlock` elements as items (typically, you'll bind the `ListBox` to a collection, as you'll learn in chapter 9).

Listing 4.8 XAML with nested elements

```
<Page
  x:Class="XamlExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:XamlExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid>
    <TextBlock Foreground="Black"
               Text="First TextBlock" />
    <ListBox Foreground="Black"
             Margin="40">
      <ListBox.Items>
        <TextBlock Text="Item 1" />
        <TextBlock Text="Item 2" />
        <TextBlock x:Name="Item3"
                   Text="Item 3" />
      </ListBox.Items>
    </ListBox>
    <TextBlock Foreground="Black"
               Text="Second TextBlock"
               Margin="20"/>
  </Grid>
</Page>
```

ListBox items

The third item in the `ListBox` was named for use in an upcoming example. Figure 4.5 shows the object tree that corresponds to the XAML in listing 4.8. The `ListBoxItem`s are automatically created by the `ListBox`, one per item.

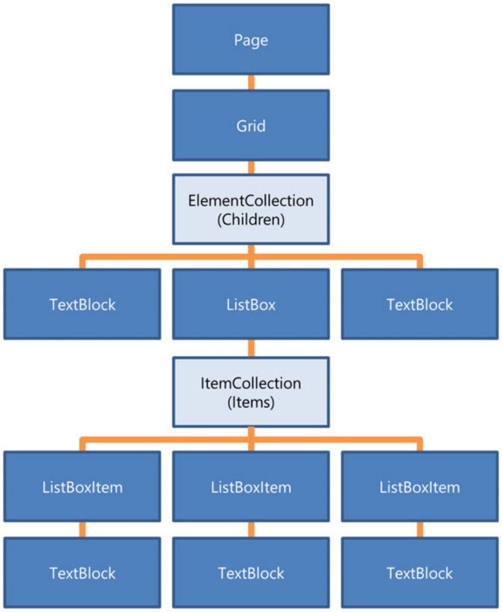

Page
Grid
ElementCollection (Children)
TextBlock
ListBox
TextBlock
ItemCollection (Items)
ListBoxItem
ListBoxItem
ListBoxItem
TextBlock
TextBlock
TextBlock

Figure 4.5 A simplified object tree showing not only the visual elements such as `TextBlock`s and `ListBox`es but also the internal collections used to contain child elements

Each element has the concept of a parent (the containing element) and may have a child or children in panel-type collection properties, content properties, or other general-purpose properties.

The *visual tree* is a filtered view of the object tree. Whereas the object tree contains all types regardless of whether they participate in rendering (collections, for example), the visual tree contains only those objects with a visual representation. Figure 4.6 shows the visual tree corresponding to the object tree in figure 4.5; note the lack of nonvisual objects such as collections.

Both of these trees have been simplified for publication. `ListBox`, for example, includes many other elements like `Border` and `ScrollViewer`. If I had created the actual tree, you'd need a special fanfold insert to see everything. Sadly, my publisher declined my request for posters and popup inserts and refused to answer me on the "scratch-n-sniff" topic.

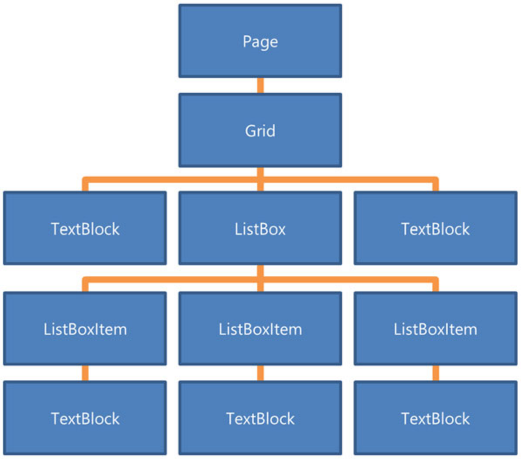

Page
Grid
TextBlock
ListBox
TextBlock
ListBoxItem
ListBoxItem
ListBoxItem
TextBlock
TextBlock
TextBlock

Figure 4.6 The simplified visual tree representation of the object tree from figure 4.5. Note that only visual elements, not collections, are represented.

WALKING THE VISUAL TREE

WinRT includes the `VisualTreeHelper` static class to assist in examining the visual tree. Using the `GetChild` and `GetChildrenCount` methods, you can recursively walk the tree from any element down as deeply as you want. The `GetParent` method allows you to trace the tree from a given element up to the visual tree root.

Taking listing 4.8's XAML as input, the code-behind to process it is shown in the following listing.

Listing 4.9 The code-behind that does the actual processing

```
public MainPage()
{
  this.InitializeComponent();
  Loaded += MainPage_Loaded;
}
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
  DependencyObject o = Item3;
  while (o != null)
  {
    Debug.WriteLine(o.GetType().ToString());

    o = VisualTreeHelper.GetParent(o);
  }
}
```

Stop when at root

You start the tree walk in the `Loaded` event handler because the tree isn't valid until the `UserControl` has been loaded. You know the walk is complete when you hit an element with a null parent—the root of the tree.

Here's the list of elements the tree-walking exercise produced:

```
Windows.UI.Xaml.Controls.TextBlock
Windows.UI.Xaml.Controls.ContentPresenter
Windows.UI.Xaml.Controls.Grid
Windows.UI.Xaml.Controls.Border
Windows.UI.Xaml.Controls.ListBoxItem
Windows.UI.Xaml.Controls.VirtualizingStackPanel
Windows.UI.Xaml.Controls.ItemsPresenter
Windows.UI.Xaml.Controls.ScrollContentPresenter
Windows.UI.Xaml.Controls.Grid
Windows.UI.Xaml.Controls.Border
Windows.UI.Xaml.Controls.ScrollViewer
Windows.UI.Xaml.Controls.Border
Windows.UI.Xaml.Controls.ListBox
Windows.UI.Xaml.Controls.Grid
XamlExample.MainPage
Windows.UI.Xaml.Controls.ContentPresenter
Windows.UI.Xaml.Controls.Border
Windows.UI.Xaml.Controls.Frame
Windows.UI.Xaml.Controls.Border
Windows.UI.Xaml.Controls.ScrollContentPresenter
Windows.UI.Xaml.Controls.Grid
Windows.UI.Xaml.Controls.Border
Windows.UI.Xaml.Controls.ScrollViewer
```

You can see that's a fair bit of stuff for such a small XAML listing. The reason behind this is that many elements have templates (chapter 9), which themselves are made up of many other elements. For example, the `ListBoxItem` contains a `Border`, which

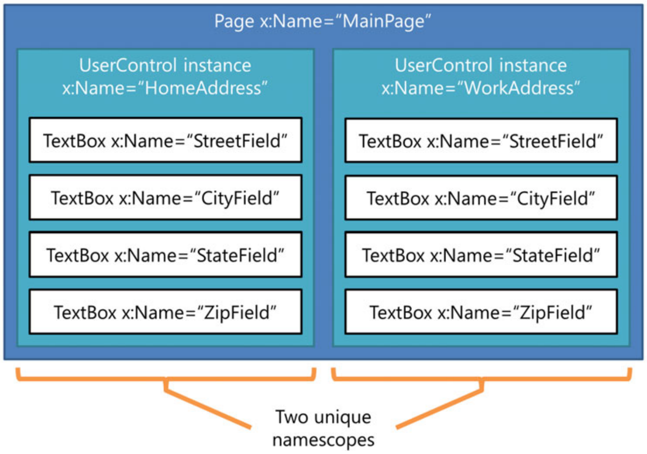

Page x:Name="MainPage"
UserControl instance x:Name="HomeAddress"
UserControl instance x:Name="WorkAddress"
TextBox x:Name="StreetField"
TextBox x:Name="CityField"
TextBox x:Name="StateField"
TextBox x:Name="ZipField"
TextBox x:Name="StreetField"
TextBox x:Name="CityField"
TextBox x:Name="StateField"
TextBox x:Name="ZipField"
Two unique namescopes

Figure 4.7 Namescope allows you to have multiple controls in the visual tree, each with the same name. This is similar to bracket-level scoping in languages like C.

contains a `Grid`, which contains a `ContentPresenter`, which contains the actual content: the `TextBlock` we started with.

You may also notice that, when you look at an object tree for an entire application, you'll have multiple instances of controls, each of which contains elements with the same name. For example, each button may theoretically contain a `Border` named `InnerBorder`. Namescope, the next topic, is how WinRT XAML ensures that the names remain uniquely addressable across the breadth of the object tree.

4.3.2 Namescope

Earlier in this chapter you saw that you can define an `x:Name` for elements in XAML. This provides a way to find the control via code and perform operations on it or handle its events.

Consider the idea of having multiple controls on the same page, each of which contains named elements, like that shown in figure 4.7. In the visual tree, you'd have two instances each of `StreetField`, `CityField`, `StateField`, and `ZipField`. But every item in the tree requires a unique name (or no name), so that can't possibly work…can it?

To handle this situation, XAML introduces the concept of a *namescope*. A namescope simply ensures that the names across instances of controls don't collide. This is similar in concept to the approach taken by ASP.NET to mangle control names to ensure they remain unique.

Our next listing shows an example control that we'll instantiate several times.

Listing 4.10 Without namescope, the button's name would be duplicated in the tree

```
<UserControl
  x:Class="XamlExample.MyNestedControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:XamlExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  d:DesignHeight="300"
  d:DesignWidth="400">

  <Grid Background="White">
    <Button x:Name="ButtonInTheControl" />
  </Grid>
</UserControl>
```

Named button

Notice how the control includes a button named `ButtonInTheControl`. If it contained more than one instance of that control, named the same way, you'd get a compile time error. What about using it three times in another control, as shown in figure 4.8?

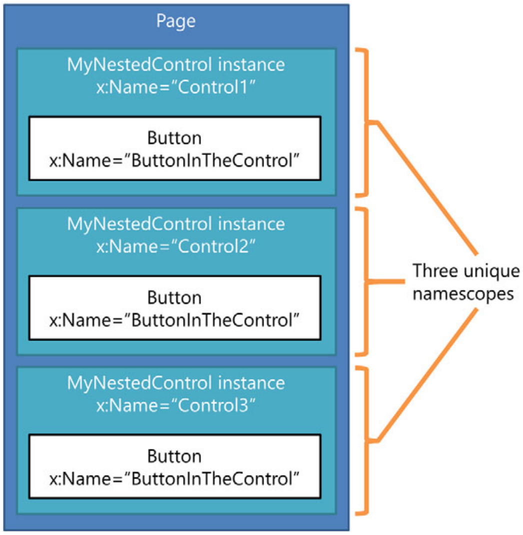

Page
MyNestedControl instance
x:Name="Control1"
Button
x:Name="ButtonInTheControl"
MyNestedControl instance
x:Name="Control2"
Button
x:Name="ButtonInTheControl"
MyNestedControl instance
x:Name="Control3"
Button
x:Name="ButtonInTheControl"
Three unique
namescopes

Figure 4.8 Another illustration of namescope in action. Each instance of `MyNestedControl` includes a button named `ButtonInTheControl`. (Note that I left out the `StackPanel` for brevity.)

The following listing shows the use of this control. Namescope is required to prevent duplicate control names.

Listing 4.11 Code that uses multiple instances of the example control

```
<Page x:Class="XamlExample.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:XamlExample"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

  <StackPanel  Background="White">
    <local:MyNestedControl x:Name="Control1" />
    <local:MyNestedControl x:Name="Control2" />
    <local:MyNestedControl x:Name="Control3" />
  </StackPanel>

</Page>
```

Multiple Instances

With three instances of the user control in listing 4.11, how does the XAML parser prevent naming collisions between all the `ButtonInTheControl` instances in the object tree but still allow you to uniquely reference each one? Namescope.

As you'd expect, using the same name twice within the same XAML namescope will result in a parsing error. This is similar to the compile-time error you'd receive if you gave two variables the same name within the same scope level in a C# application. If the two names are in different namescopes (like different instances of user controls), then there are no problems.

This is one of those "Duh! Of course it works that way" implementation details. But it shows you what the team had to implement to make the seemingly obvious things behave as you'd expect. In practice, you typically don't need to worry about namescopes unless you're loading and parsing XAML at runtime using the XamlReader.Load API (in which case it can be very important). The namescopes are created for you automatically at runtime when you instantiate your controls.

For further reading on XAML namescope, see MSDN: http://bit.ly/WinRTXamlNamescope.

Understanding both object trees and namescope provides important insight into what the XAML parser is doing on your behalf. For example, knowing the size of an object tree can be useful when performance tuning your application, because the more elements in your tree, the harder the runtime has to work.

4.4 Summary

The intent of this chapter was to familiarize you with XAML. That's actually a difficult task to fit into a single chapter because XAML is simply the markup manifestation of WinRT and .NET objects. In truth, most of this book is about different aspects of XAML and different types you can represent in it.

For those reasons, this chapter covered the core XAML concepts you need to understand before you can make sense of anything else. Elements and properties are absolutely essential because they're the meat and potatoes (or tofu and broccoli) of markup.

You also learned how the property system used in XAML is a bit different from the one you may have used in C# in your own classes and in other presentation technologies. Dependency properties and attached properties are key to supporting the binding and animation systems and important in producing panels with flexible layout.

Finally, an understanding of the object trees created by the XAML parser will provide you with important insight into how heavy a UI you're making and how hard you're making WinRT work.

XAML is my favorite presentation technology. It has the strong structure of XML with the flexibility of WinRT and .NET and none of the baggage of other presentation markup languages like HTML. It was purpose-built for this use, and it shows. It's certainly not perfect (no, okay, it's perfect), but I've enjoyed working in it for the better part of a decade now. I think you will too.

5

Layout

This chapter covers

- The layout system
- Alignment, margins, and padding
- Layout rounding
- Performance considerations

Layout systems across different technologies vary greatly in complexity. Take, for example, the Windows Forms layout system. Fundamentally, that layout system involves absolute X and Y coordinate pairs and an explicit or implicit Z order. Controls can overlap, get clipped (cut off or cropped) on the edge of the window, or even get obscured. The algorithm is pretty simple—sort by Z order (distance from the viewer) and then transfer the pixels to the screen.

For another example, look to HTML and CSS. HTML and CSS support elements that must size to content and page constraints (tables, divs) as well as support absolute positioning, overlapping, and so forth. It's more of a fluid approach, where the size and position of one element can affect the size and position of another. Therefore, the layout system for HTML and CSS is significantly more complex than that for something like Windows Forms.

Like HTML, XAML supports both types of layout: content that self-sizes based on constraints, and content that's simply positioned by way of an X and Y coordinate pair. Depending on the container in use, it can even handle laying out elements on curves or radially from a central point. The complexity that makes this flexible layout system possible deserves a deeper look.

We'll first dive into the multipass layout system so you have a solid grounding for what to expect out of the panels and other elements involved in layout. Next, we'll move on to those properties that govern layout in most situations: alignment, padding, and margins. After that, we'll take a brief look at layout rounding, something commonly used to ensure crisp lines and corners in our UI—a must for Modern Style apps. The chapter wraps up with a discussion of performance considerations for apps of all types.

5.1 Multipass layout—measuring and arranging

Like so many things, you can easily create your own first Windows Modern Style apps without understanding the multipass layout system. Many developers have gotten far into developing with XAML (WPF, Silverlight, and Windows) without ever hearing about layout. The first time they see a "layout cycle detected" or similar error, however, it stops them in their tracks.

Details are important, and few details are as vital as the mechanism by which elements are positioned and sized onscreen. This system, the multipass layout system, is the focus of this section.

Just as with WPF and Silverlight, layout in WinRT XAML involves two primary passes: *measure* and *arrange*. In the measure pass, the layout system asks each element to provide its ideal dimensions given a provided maximum size. In the arrange pass, the layout system tells each element its final size and requests that it lay itself out and also lay out its child elements. A full run of measuring and arranging is called a *layout pass*, depicted (along with the render step) in figure 5.1.

Layout information is retained from frame to frame and is recalculated only when necessary. Typically this comes from animating a layout-related property (a

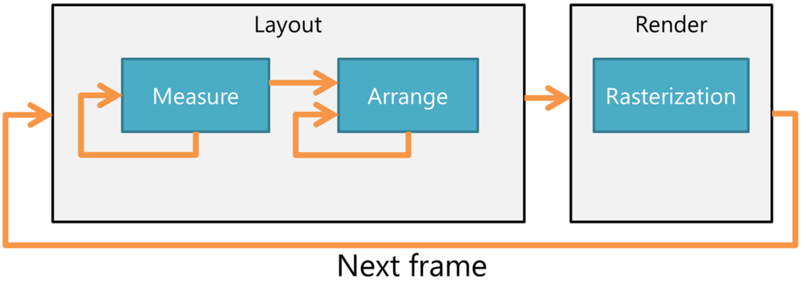


Figure 5.1 The layout and rendering process for XAML application. Measure and layout happen for the visual tree, and rasterization for the frame (or cached layers within the frame). Note that this diagram doesn't get into caching (which breaks this up into multiple layout/render pairs), 3D composition with C++, or other substeps.

general no-no) or by otherwise changing layout properties or introducing new elements to the tree.

You can imagine that layout and render can be quite a bit of work to accomplish at a reasonable frame rate, especially as your visual tree becomes more complex. Toward the end of this chapter, I'll offer up some pointers to help with performance in this area.

In this section, we'll look at the layout process. First, we'll look at the measure pass, followed logically by the arrange pass. With those covered, we'll also look at the `LayoutInformation` class, something that's helpful for creating your own panels and also good to understand to get insight into the layout system.

5.1.1 The measure pass

Whenever elements need to be rendered to screen, the layout system is invoked for an asynchronous layout pass. The first step in layout is to measure the elements. On a `FrameworkElement`, the measure pass is implemented inside the virtual `MeasureOverride` function, called recursively on the visual tree (the tree of all visual elements):

```
protected virtual Size MeasureOverride(Size availableSize)
```

The `availableSize` parameter (a `Windows.Foundation.Size` type) contains the maximum amount of space available for this object to give to itself and its child objects. If the `FrameworkElement` is to size to whatever content it has without any initial constraints, the `availableSize` will be `double.PositiveInfinity`.

The function is responsible for returning the size the element requires based on any constraints or sizes of child objects.

Note that `MeasureOverride` isn't called directly from the layout system; it's a protected function. Instead, this function is called from the `UIElement`'s `Measure` function, which, in turn, is called by the layout system.

At the end of the measure pass, each element will have a `DesiredSize`, which represents the size it wants to be. This information is then used as input into the arrange pass.

5.1.2 The arrange pass

The second pass of layout is to arrange the elements given their final sizes. On a `FrameworkElement`, the arrange functionality is implemented inside the virtual `ArrangeOverride` function, also called recursively:

```
protected virtual Size ArrangeOverride(Size finalSize)
```

The `finalSize` parameter contains the size (the area within the parent) this object should use to arrange itself and child objects. The returned size must be the size actually used by the element and less than or equal to the `finalSize` passed in; larger sizes typically result in clipping by the parent.

Similar to the relationship between the measure pass and `MeasureOverride`, `ArrangeOverride` isn't called directly by the layout system. Instead, the `Arrange` method on `UIElement` is called, which then calls the protected `ArrangeOverride` function.

At the end of the arrange pass, Windows has everything it needs to properly position and size each element in the visual tree.

Hooking into the rendering process

Once layout is completed, everything goes into the rendering pipeline. There, the system goes through a number of steps to composite the different pieces, bring in cached layers, and more.

In Silverlight, it was often important to have a deep understanding of the rendering pipeline in order to optimize performance for frame-based games and other high-performance media. In fact, I dedicated something like 10 pages to covering this topic in *Silverlight 5 in Action*.

The availability of native code and access to DirectX has really changed the approach you'll want to use, making this detailed information less necessary than it once was. In Windows Modern Style apps, the recommended approach for handling highly performance-sensitive scenarios is to use C++ and DirectX. You can either do this for the entire application or simply write critical rendering code in C++/DirectX and surface it as a WinRT extension assembly you can then use from your C# (or HTML/JS) app.

Nevertheless, there are times when you may want to hook into the rendering process to do something for each frame that's rendering. That may be as simple as keeping a count of frames rendered, swapping a back buffer to simulate an immediate-mode rendering system (again, C++ is better here), or performing game loop-style operations. For those situations, you can use the `CompositionTarget.Rendering` event.

```
public MainPage()
{
  this.InitializeComponent();
  CompositionTarget.Rendering += OnRendering;
}
void OnRendering(object sender, object e)
{
  RenderingEventArgs args = e as RenderingEventArgs;
  Debug.WriteLine(args.RenderingTime.ToString());
}
```

Note the cast to `RenderingEventArgs` in this code. This is pretty unusual and not something you'd figure out without knowing something about the underlying code. The underlying code is actually sending an instance of `RenderingEventArgs`, but the event signature is just a regular object.

There's no guarantee that the callback will happen at the max frame rate. Although it often does work out this way, many factors, including the amount of work being done inside the callback and the overall speed of the system, contribute to how often this runs. You can generally expect the callback to happen once per frame, assuming your code is well behaved.

Measure and arrange together make up the layout process, but they aren't something you're likely to interact with directly unless you're building custom controls and panels.

Nevertheless, there are times when you want to get some runtime insight into what's going on in the process. The `LayoutInformation` class can provide some of that.

5.1.3 The LayoutInformation class

The `LayoutInfomation` class in `Windows.UI.Xaml.Controls.Primitives` contains a few methods that are useful to folks implementing their own `MeasureOverride` and `ArrangeOverride` code. Specifically, `GetLayoutSlot` is helpful when hosting child elements in a custom panel, and `GetExceptionElement` can help provide details about errors during the layout process.

GETLAYOUTSLOT

Regardless of its actual shape, each visual element in XAML must be placed into a layout slot in a panel. Figure 5.2 shows the relationship between a layout slot and the child element hosted in a panel.

The layout slot is the maximum size to be used when displaying an element. Portions of the element that fall outside the slot will be clipped, or cut off. In the case of a panel like the `Grid` (next chapter), the element will typically be resized to fit inside the layout slot, taking into account the element's margins and alignment properties. For this reason, the layout slot may be a different size than the element's logical bounding box, although the two terms are often used interchangeably.

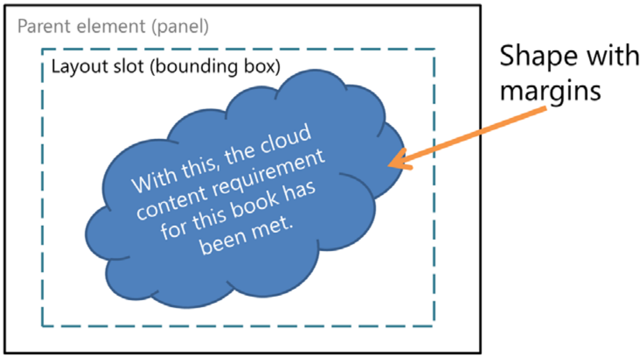

Parent element (panel)
Layout slot (bounding box)
With this, the cloud content requirement for this book has been met.
Shape with margins

Figure 5.2 The relationship between the layout slot and the child element in the slot

To get the coordinate values for the layout slot for an element, you can call the static function `GetLayoutSlot`:

```
public static Rect GetLayoutSlot(FrameworkElement element)
```

The returned `Windows.Foundation.Rect` will contain the layout slot for that element. This return value can be useful when creating a custom panel or when debugging layout issues.

GETLAYOUTEXCEPTION

Speaking of debugging, sometimes we have bugs in our code—sometimes (Gasp!) so does Microsoft.[1] An exception during a layout pass can be really difficult to track down, because it's not always obvious where it came from or what exactly was being processed at the time.

---

[1] With that comment, my career plan just changed to "Sell Girl Scout cookies outside the supermarket."

The `GetLayoutException` function, which can be used in an app-wide global error handler if you'd like, returns the element that was in layout when the unhandled exception occurred.

```
public static UIElement GetLayoutExceptionElement(object dispatcher)
```

You must provide this method with the dispatcher from your UI level. The `Dispatcher` is the class that makes it possible for code running on a background thread to send (or dispatch) calls to the UI thread. This is necessary because only code on the UI thread can modify controls.

In WinRT XAML, the type here is `object`, but it's really expecting a `Dispatcher`. Presumably it was written this way because of some oddness around circular dependencies between different parts of WinRT, but your guess is as good as mine. In any case, you can get the correct `Dispatcher` by using the `Dispatcher` property of any object in your visual tree. The layout process is one of the most interesting things about XAML and is helpful to learn, especially for debugging. It's a recursive process that makes two passes across the visual tree for your application. For that reason, the depth and complexity of your visual tree have a direct impact on layout performance. Toward the end of this chapter, we'll look at ways to optimize performance. Before that, however, it's important to understand a few other key properties that directly factor into layout.

5.2 UIElement layout properties

If layout in XAML was as simple as providing X and Y coordinates for each element, plus a height and width, this chapter would be a page long. Instead, as you've seen so far, layout in XAML is a complex process involving panels and content controls and their child elements. Each panel is responsible for laying out its children, in whatever way that panel implements. Because of this, it's relatively easy to create custom layout panels that position elements in ways the product teams didn't have time to implement—or perhaps never even considered.

In order for layout to be at least a little sane, you need some common ways to provide information to the layout system. There are a number of element properties that affect layout. Most of those properties are defined on the `UIElement`, whereas others are widespread enough that they probably should be.

As you'll see in the next chapter when we discuss `Grid`, `StackPanel`, and others, several of the common panels respect the margins and alignments of their children.

In this section, we'll start with the most basic properties: `Width` and `Height`. Whenever you consider sizing an element, these are probably the first that come to mind. You'll learn, however, that they aren't always necessary or desired. Next, we have the `HorizontalAlignment` and `VerticalAlignment` properties. These, alongside the `Padding` and `Margin` properties also covered here, will be the most common way of sizing and positioning elements in XAML. Finally, sometimes we don't want the layout system to be so flexible, so we have a tweak property, `UseLayoutRounding`, which enables us to snap elements to pixels.

5.2.1 *Width and Height, plus ActualWidth and ActualHeight*

The most basic properties are `Width` and `Height`. `Width` is the horizontal size, and `Height` is the vertical size, both expressed in logical pixels (pixels defined by the Windows current screen resolution and DPI [dots per inch] settings, not necessarily by actual device pixels). Every element that you place on a panel has these properties, so you can easily set the dimensions. Before you do that, however, consider that you're hardcoding dimensions in a system that was designed from the ground up to be flexible enough to easily support different resolutions with automatic resizing and, if you use a `ViewBox` or similar, automatic rescaling.

ViewBox and render transforms

A `ViewBox` is a UI element that automatically scales its contents using a render transform. For vector content, this gives really clean scaling to support showing the same content at higher or lower resolutions. The child element of a `ViewBox` (typically a `Grid`) must have a set `Width` and `Height` so the `ViewBox` has a size to start from.

```
<ViewBox>
  <Grid Width="1024" Height="768">
    ... content to scale.
        Size to initial 1024x768 height to start ...
  </Grid>
</ViewBox>
```

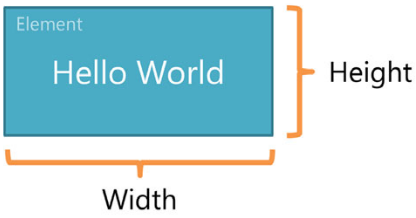
To scale the content, put the `ViewBox` in a `Grid` with alignment set to `Stretch` in both directions, or manually set the size of the `ViewBox` using `Width` and `Height` properties.

A render transform is a way of manipulating an element's size, location, rotation, or orientation (including a 3D-like projection) after layout has been calculated.

```
<TextBlock FontSize="42" Text="This Text will Scale">
  <TextBlock.RenderTransform>
    <ScaleTransform ScaleX="3.0" ScaleY="1.5" />
  </TextBlock.RenderTransform>
</TextBlock>
```

Because render transforms happen after layout, using them doesn't impact layout in any way—a potentially huge performance boost for moving, rotating, and resizing elements. The trade-off is that render transforms don't impact layout: They can't cause the control to take up additional space in a `Grid` or move subsequent elements in a `StackPanel`. We'll apply render transforms, specifically a rotate transform, in chapter 21.

For more information on render transforms, please see MSDN: http://bit.ly/WinRTRenderTransform.



In fact, most of my UIs are made up primarily of elements where there is no `Height` or `Width` specified. That's not to say those properties are evil or anything, but if you have to set them to get something to work correctly, you may find that you actually have some other layout sizing issue that needs to be solved. As long as you realize that you sacrifice some resizing flexibility when setting `Height` and `Width`, go ahead and use them.

To use the properties, set them as shown here:

```
<TextBlock Text="Hello World"
           Height="100" Width="250" />
```

ACTUALHEIGHT AND ACTUALWIDTH

During layout, the panel that owns the element can modify the size of the element if necessary. When there's a `Height` and `Width` specified, this usually won't happen, but it can. In those cases, the element's real size can be found in the `ActualHeight` and `ActualWidth` properties.

If you don't explicitly set the `Height` and `Width` properties of a control, the `ActualHeight` and `ActualWidth` properties may be zero or not a number (`double.NaN`). Why is that? Because of the asynchronous nature of the layout pass, `ActualHeight` and `ActualWidth` might not be set at any specific point in time from run to run or, more importantly, might actually change their values over time as the result of layout operations.

`ActualHeight` and `ActualWidth` are set once layout is complete and may also be affected by layout rounding settings or content. In short, check them, and if they're zero or `NaN`, they haven't been set.

LAYOUTUPDATED EVENT

If you want a single place where you can guarantee they'll have a value, subscribe to the `LayoutUpdated` event on the element and check them there.

Despite the name, the `FrameworkElement`'s `LayoutUpdated` event isn't technically part of the layout pass. Instead, it's fired as the last event before an element is ready to accept input. `LayoutUpdated` is the safe location for inspecting the actual size and position of the element or otherwise responding to changes in same.

Don't do anything in `LayoutUpdated` that would cause another layout pass. For example, don't change the size or position of an element, modify its contents, change its layout rounding, or otherwise manipulate properties that could change the size of the element's bounding box. If you have multiple nested layout passes and they take longer than the time allowed for that frame, the Windows XAML engine may skip frames or throw a layout exception

I've offered so many cautions against relying on `Height` and `Width` because I'm a much bigger fan of using `HorizontalAlignment` and `VerticalAlignment` properties along with `Margin` and `Padding`.

5.2.2 Horizontal and vertical alignment

Okay, so I spent the whole last section warning you away from using `Height` and `Width` as though you might get some strange communicable disease just by thinking in terms of hardcoded pixel values.

Really, what I wanted is for you to keep an open mind, because the real stars of layout are coming in this section and the next. First, I'd like to introduce you to my good friends `HorizontalAlignment` and `VerticalAlignment`. Figure 5.3 shows them in all their glory.

In the top left, you can see that a `HorizontalAlignment` of `Left` aligns an element to the left. Similarly, `Center` and `Right` align elements to the center and right, respectively. In order to get them to align, you'll need to give the element a width value.[2] The same applies for `VerticalAlignment` with `Top`, `Center`, and `Bottom` values. In those cases, you'll need to provide the elements with a `Height` value in order for the alignment to make sense.

To get an element to align at the top left of the container, with 10 px of spacing around it, you'd use the following markup inside a `Grid`:

```
<TextBlock Text="Hello World"
           HorizontalAlignment="Left"
           VerticalAlignment="Top"
           Margin="10"/>
```

It just so happens that `TextBlock` is an element that works well with no `Height` or `Width` specified. In fact, it's rare for developers to provide an explicit size to `TextBlock` elements, because their text usually comes from binding to some model object or resource.

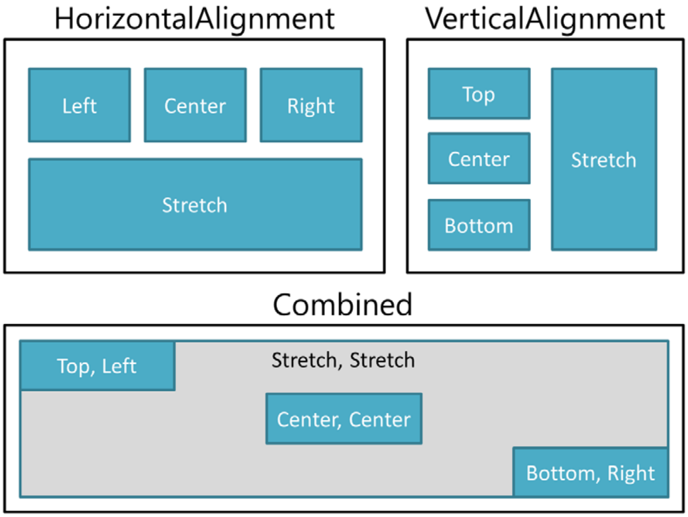


Figure 5.3 An illustration of alignment values. In the bottom, combined set, you can see both horizontal and vertical alignment combined, including a rectangle in the background with `Stretch` for both of the alignment values.

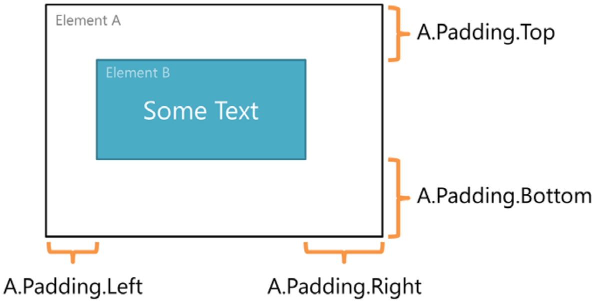
---

[2] That is not backpedaling, I swear!

Even more interesting to me are the `Stretch` values. When you place elements in a `Grid` panel—the single-most-popular panel, and covered in the next chapter—`Stretch` is the default for both `HorizontalAlignment` and `VerticalAlignment`. `Stretch` simply says "take up all the available space." When you combine `Stretch` with the `Margin` property we'll cover shortly, you can create a flexible UI that automatically resizes to the available space provided.

`Padding` and `Margin` go hand-in-hand with the alignment properties, so let's look at them now.

5.2.3 Padding



Okay, I lied a little in titling this section: `Padding` isn't a `UIElement` property. `Padding` is defined on `Control` and several other higher-level classes like `TextBlock`. But it has such a tight affinity with `Margin`, and with the layout process itself, that I find it appropriate to cover here.

Padding is the open space an element (a panel or content control) provides around its contained child(ren). In XAML, the `Padding` value is typically expressed as a space or comma-delimited `Thickness` type like this:

```
<ContentPresenter x:Name="A" Padding="8,10,20,25"/>
```

The values are specified clockwise from the left side—`Left`, `Top`, `Right`, `Bottom`—and may be whole integers or floating-point values. In this case, the left margin is 8, top margin is 10, right is 20, and bottom is 25. If you prefer, you can delimit the values using spaces, like this:

```
<ContentPresenter x:Name="A" Padding="8 10 20 25"/>
```

To share the same padding on all four sides, simply specify the single value, like this:

```
<ContentPresenter x:Name="A" Padding="10"/>
```

In this example, all four sides will have a padding value of 10. Finally, to set the value from code, don't manipulate the properties directly, but instead assign a new `Thickness` to the `Padding` property, like this:

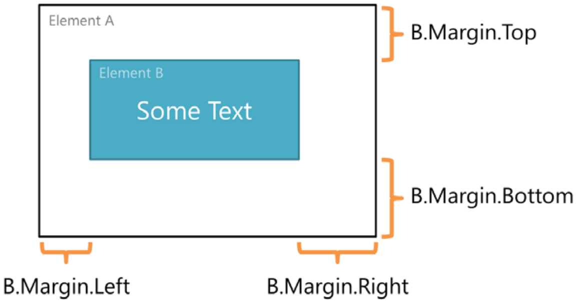
```
SomeElement.Padding = new Thickness(10);
SomeElement.Padding = new Thickness(8, 10, 20, 25);
```

The first line sets the padding to a uniform thickness of 10. The second sets the four individual values just as seen in the markup. Because `Thickness` is a structure and doesn't handle any property change notifications, you must assign the entire structure, not the individual `Left`, `Top`, `Right`, and `Bottom` properties. You'll get an exception if you manipulate the individual properties.

The `Margin` property is very similar to `Padding`, but instead of controlling space on the inside of certain controls, it controls the buffer space around the outside of just about any element.

5.2.4 Margins

The `Margin` property of an element represents the empty space around the element. In use, it's far more common to set the `Margin` property than the `Padding` property. In XAML, the `Margin` is typically expressed as a space or comma-delimited `Thickness` type like this:



```
<Rectangle x:Name="B" Margin="8,10,20,25"/>
```

Just as with `Padding`, the values are specified clockwise from the left: `Left`, `Top`, `Right`, `Bottom`. They may be whole integers or floating-point values. In this case, the left margin is 8, top margin is 10, right is 20, and bottom is 25.

To share the same margin on all four sides, simply specify the single value:

```
<Rectangle x:Name="B" Margin="8"/>
```

In this example, all four sides will have a consistent margin of 8. Just as with `Padding`, you can set the value from code using either a single value or the individual `Left`, `Top`, `Right`, and `Bottom` values in the `Thickness` constructor. You must also assign the entire structure at once, just as with `Padding`.

XAML provides both `Margin` and `Padding` properties because sometimes you own the container, and sometimes you own the contained element. Having both means that regardless of which element you're creating (or styling), you can control the whitespace to provide a little breathing room around the controls.

Margins and padding generally work in concert with `HorizontalAlignment` or `VerticalAlignment` properties: Left margins only come into play when the control has `HorizontalAlignment` equal to `Left`, `Center`, or `Stretch`. Similarly, right margins only come into play for `Right`, `Center`, or `Stretch` `HorizontalAlignment` values. Padding works the same way depending upon the alignment values of the contained child control (or in the case of a `TextBlock`, the contained text).

It's also possible to defeat margins by providing explicit values to the `Width` or `Height` of different controls. A common layout trick is to provide just one of the dimensions, for example, `Height`, and then provide a `HorizontalAlignment` value of `Stretch` and a `VerticalAlignment` of `Top`. That will let you have a bar across the top of the screen, with a known height, but with a width that depends on the width of the container (a page in this case)—think toolbar, banner, or top AppBar.

This can all get a bit muddy if you start providing too much help to the layout system. If you find that controls are not sizing as you'd expect, start by removing `Width` and `Height` values, and pare back from there.

`Width`, `Height`, `HorizontalAlignment`, `VerticalAlignment`, `Margin`, and `Padding` are all key inputs into element layout. Several of them affect size, several affect position, and most of them can affect both in specific situations.

In general, if you can get away with it, it's better to use the two alignment properties along with the margin in order to control size and positioning of an element. In panels such as the `Grid` (which we'll cover in the next chapter), this makes it simple to adjust the interface for different display resolutions or container sizes. But if you really do need to have an absolute size for an element, `Width` and `Height` will get you there.

All of the numeric properties discussed so far (`Width`, `Height`, `Margin`, `Padding`) support using floating-point values (double-precision decimals like the number 12.375) for the value. The result of that can be shapes that don't start or stop on physical pixel boundaries. Sometimes, you want to have better control over that. At those times, you can use *layout rounding*.

5.3 Layout rounding

Like Silverlight and WPF, WinRT XAML supports aligning elements on subpixel boundaries. This means that you can actually position something at a coordinate like X = 2.718 and Y = 3.1415. This is especially useful for smooth animation and smooth scaling. An unfortunate side effect of this, however, is the loss of crisp lines when positions fall outside integer pixel values. In the Modern Style aesthetic, this can be especially disappointing: Sometimes, you really want that 1 px line to be just 1 px thick and not antialiased to 2 px in thickness.

One simple way to avoid this problem is to place your elements on whole pixel locations. But when your element is nested inside a panel, which is inside a control, which is in a stack panel located in another grid—all of which can have margins, padding, and other properties affecting layout—you can't easily calculate exactly where your element will appear.

Figure 5.4 shows two rectangles aligned on subpixel boundaries, one with layout rounding on and one with it off.

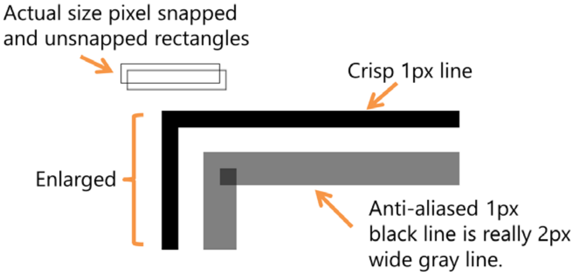


Figure 5.4 Two rectangles shown actual size and enlarged. One rectangle uses layout rounding and is snapped to device pixels. The second does not and has a wider and fuzzier appearance.

WinRT XAML supports a property of the `UIElement` called `UseLayoutRounding`. When `UseLayoutRounding` is set to `True` (the default), the layout system will round the points of your element to the nearest whole pixel. In this case, elements are said to be “snapped” to pixels. When set to `False`, Windows will respect the subpixel location of the points and won’t attempt to move them; this can be especially useful for animation. The following listing shows the impact of layout rounding on two rectangles. The first rectangle has layout rounding turned on; the second has it turned off.

Listing 5.1 Layout rounding in action with two rectangles

```
<Grid Background="White">
  <Rectangle Margin="10.5"
             UseLayoutRounding="True"
             HorizontalAlignment="Left"
             VerticalAlignment="Top"
             Fill="Transparent"
             Stroke="Black" StrokeThickness="1"
             Width="150" Height="30" />
  <Rectangle Margin="20.5"
             UseLayoutRounding="False"
             Fill="Transparent"
             HorizontalAlignment="Left"
             VerticalAlignment="Top"
             Stroke="Black" StrokeThickness="1"
             Width="150" Height="30" />
</Grid>
```

Rounding

No rounding

In this listing, you can see that the rectangle that isn’t rounded to the nearest pixel has lines that are actually 2 pixels thick and colored 50% gray instead of black. There’s also a little overlap in the corner that’s 75% gray. When viewed in its native 1:1 pixel resolution, it looks fuzzy. When layout rounding is turned on, the result is a crisp line with sharp corners and no fuzz. When verifying this yourself, make sure your design isn’t zoomed in or out.

`UseLayoutRounding` is respected by almost every element in XAML. The `Polygon` class exposes this property from its base class but ignores it. Polygons are expected to be complex shapes where layout rounding wouldn’t really make sense, so layout rounding is a no-op.

> **NOTE** When sharing code and markup with WPF desktop applications, it’s important to note that layout rounding is turned on by default in WinRT (as in Silverlight). This is in contrast to WPF, where it’s turned off by default.

A flexible layout system needs a number of different ways to size and position elements. In the end, most of the responsibility comes down to the panel you place the elements in. But there are a number of common properties that are used as inputs into layout.

`Height` and `Width` are the simplest of the positioning properties. I tend not to use them for many of the elements on a page because I want to take advantage of automatic resizing based on available screen real estate.

`HorizontalAlignment`, `VerticalAlignment`, `Margin`, and `Padding` work together to provide a better way to position elements while retaining the flexibility of the system. If you want to align something to an edge, provide a margin if desired and align it to the edge using one of the two alignment properties. If you want something to take up all available space, set its alignment properties to `Stretch` and optionally provide a margin.

Finally, all this flexibility can sometimes cause things to look a little fuzzy. Apparently Fuzzy Wuzzy[3] was actually an element aligned on fractional pixel boundaries. To get around this, set `UseLayoutRounding` to `True` to round an element to the nearest whole pixel value.

So far, we've talked about flexibility and the power of the layout system. To use it effectively, you'll need to understand the performance implications of decisions and simple changes you can make to help your app feel snappy.

5.4 Performance considerations

Layout is a recursive process; triggering layout on an element will trigger layout for all the children of that element, and their children, and so on. For that reason, you should try to avoid triggering layout for large visual trees as much as possible. In addition, when implementing your own `MeasureOverride` or `ArrangeOverride` code, commonly done in custom panels and highly customized controls, make sure it's as efficient as possible.

There are a number of other performance considerations you should keep in mind when designing your UI. In this section, we'll cover a few of the most important ones, including keeping the visual tree shallow, caching subtrees, UI virtualization, and sizing and positioning elements.

5.4.1 Keeping the tree shallow

When Silverlight was first designed, the button controls had a very complex layout made up of multiple rectangles overlaid, some with gradients, all with different margins, borders, opacity ramps, and more. The visual tree for a single button was actually quite complex. Before Silverlight was released, the product team cleaned up the button template and greatly simplified it to just a handful of elements.

SIMPLIFY

WinRT XAML has even simpler buttons, with no gradients. This is no accident—a complex UI with varying opacities, gradients, and just a deep tree will take longer to lay out and render than a simpler UI. This very fact is part of the reason why Modern Style UIs look the way they do—it's about the performance and experience, not unnecessary performance-killing decoration.

When designing your own UI, follow Modern UI principles not only for the aesthetic but also for the performance. Keep in mind how many elements you have in any given layout pass, and try to keep that number as small as possible. If you have a super-complex UI with tons of touch points, consider making it up from many very simple elements. In fact, you may even consider making it up of many static images rather than XAML subtrees.

---

[3] Meaning, Fuzzy Wuzzy was a bear: http://en.wikipedia.org/wiki/Fuzzy_Wuzzy_(song).

USE PNGS AND NOT COMPLEX XAML

One way to keep the tree shallow is to use PNGs instead of complex XAML for things like icons and similar elements. When to use an image instead of resizable vector content will depend greatly on what you plan to do with it: Resizable content is probably not a good candidate to be a static image, but static content sure is.

Consider using PNGs for all your custom AppBar and other icons, as well as for any background imagery your application uses.

In cases where static images won't cut it, consider caching your prerendered subtrees.

5.4.2 Caching

In XAML, caching can be used to render an entire subtree to an image buffer. This is handled by setting the `CacheMode` on an element in the tree. Everything in that element's subtree will then be cached according to the value set and then rendered as though it were a static image. For example, to enable caching on a grid, you'd set `CacheMode` to `BitmapCache` like this:

```
<Grid CacheMode="BitmapCache">
```

Like so many performance optimizations, caching is something best explored toward the end, after you've gotten everything else working and optimized. Sometimes, caching can hurt performance. How? For any layer you cache, Windows must render everything above it to one surface and everything below it to another. It can then composite the three different surfaces (lower render, cached layer, upper render) to create the final rendered output. This takes quite a bit more memory (typically GPU memory) and incurs a performance hit from the process of creating the cached images. In some cases, it can take longer to do this than to just let Windows optimize the rendering as it normally would.

It's also worth noting that animations inside a cached subtree are, as you'd expect, disabled. If you need animations to run, you'll need to cache at a lower level or not at all.

Caching renders elements to a backing bitmap, thereby reducing the number of elements in any given layout pass. There's another important type of performance optimization that takes a different approach to reducing the number of elements in the tree: virtualization.

5.4.3 Virtualization

Sometimes you can't control the number of elements in the tree. Take the example of a large list of all the photos on your machine. Paging is a really ugly solution, and delayed loading can be a bit of a pain (although you may still opt to use it in this specific case). If the performance issue is in creating and rendering all the elements, you can get around this by using UI *virtualization*.

Virtualization is the reuse of elements to display data and the caching of that rendered output. A subset of the built-in controls and panels support UI virtualization; you'll see which ones when we cover panels and later when discussing the new `GridView` and `ListView` controls. For those, precreated elements are reused with new data. The result is a reduction in the number of in-memory elements, as well as a reduction of `MeasureOverride` and `ArrangeOverride` calls.

The actual approach to enabling virtualization, and the impact of doing so, depends on the specific control you're using, so I'll defer coverage of the specifics for now.

5.4.4 Sizing and positioning

Another performance consideration has to do with sizing and positioning elements. For example, if you change the margin of an element or modify its width or height, you'll trigger a layout pass. In fact, one of the nastiest things you can do is trigger another layout pass during the current layout pass. You do want things to render at some point, after all. But, if you instead create a render transform to either move or resize that element, you won't trigger a pass. We'll cover render transforms in chapter 21 in the context of rotating a space ship.

Understanding the layout system helps take some of the mystery out of what happens when you size elements in XAML and they don't quite do what you might've expected them to do. It's also a key concept to understand if you plan to implement your own panels/container controls.

WPF has the concept of a layout transform. This type of transform is parallel to a render transform but triggers a layout pass. As you've seen here, triggering a layout pass can be an expensive operation, especially if done inside an animation. For performance considerations and because of their relatively low adoption, layout transforms were omitted from both WinRT XAML and Silverlight. The render transforms provided are almost always adequate—and often superior—to solve problems we used to solve with layout transforms.

Layout is one of the most visible things that affect how your application performs. You can, of course, still write slow code, but nothing impacts the user experience like a UI that's laggy and otherwise isn't keeping up with the user. Because of that, I included a few performance ideas in this section. The most important was to keep the tree as shallow as possible. When that's not possible, you can use caching and virtualization. Finally, it's important to realize that triggering a layout pass by changing the layout-related properties of an element obviously affects performance.

5.5 Summary

The layout system in XAML is about as sophisticated and cohesive a system as I've seen. It offers great power and flexibility while not getting in your way or filling elements with scenario-specific layout properties.

The multipass layout process recursively traverses the tree measuring all the elements and then does the same to arrange the elements with their final size. This multipass approach allows panels to size their children across the entire visual tree.

Key inputs into the layout process are the height and width of elements. Sometimes those sizes are explicitly provided through the `Height` and `Width` properties, and other times they come from values derived through the parent's `Padding` property, the element's `Margin` property, and `HorizontalAlignment` and `VerticalAlignment` properties. Finally, the size of the element can be affected by the value of `UseLayoutRounding`, a property that enables snapping the element to device pixels in order to maintain crisp and clear lines.

Finally, with all this flexibility comes the possibility of overdoing it and having less than stellar performance. There are a number of ways to tweak the system to provide for better performance, but by far the most effective is to simply give the system fewer elements to lay out. You can do this by simplifying the UI, breaking the UI into multiple pages, caching layers of the tree to static images, or virtualizing large lists of data. Of course, you'll also want to avoid triggering unnecessary layout cycles by changing the size or location of elements using layout properties rather than render transformations.

In the next chapter, we'll take this layout information and put it into practice using the different panels in WinRT XAML.

6

Panels

This chapter covers

- The `Panel`, `Canvas`, and `Grid`
- The `StackPanel` and `VirtualizingStackPanel`
- Creating a panel from scratch

In the previous chapter you learned about the WinRT XAML layout system. This system is based on the `UIElement` class, the `FrameworkElement` class, and, most importantly, the `Panel` base class.

This chapter is about the classes that derive from the `Panel` base class. Collectively, these are referred to simply as *panels*. A panel in XAML contains any number of child elements in the `Children` collection. How it arranges those children is up to the panel itself. Some panels use simple X,Y positioning, some use a row and column approach, and some simply automatically arrange based on the order in which elements are added.

Panels are vital to working in XAML. Before you place your first `Button`, or `TextBlock`, or other control, you'll start with a panel. A solid understanding of how each panel is intended to be used will save you a great deal of debugging later. I can't tell you how many times I've seen developers struggling with a layout in nested panels where some element just won't position itself properly. In just about

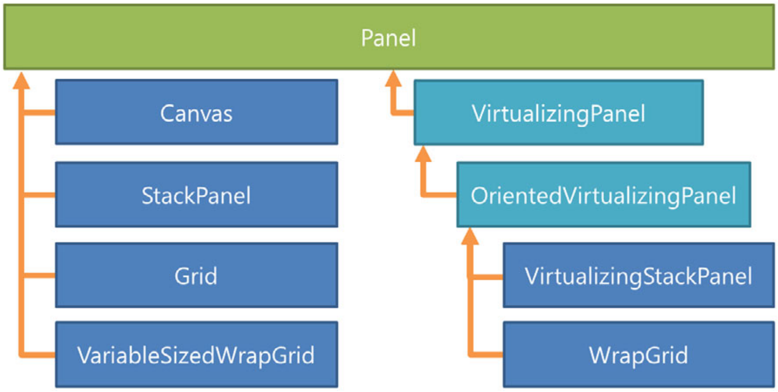


Figure 6.1 The `Panel` class hierarchy. All panels must ultimately derive from the `Panel` class (which itself derives from `FrameworkElement`). Although they are panels, they can't be used directly; they must be used only within `ItemsControl` templates such as the `GridView` and `ListView` covered in chapter 10.

every case, it has been because the developer tried to force a panel to do something another panel was better suited for. Luckily, because panels all have a common API for adding children, swapping out one panel for another is relatively easy.

The built-in panels, shown in figure 6.1, cover the vast majority of common layout scenarios. But should you find a scenario that you can't easily support with these panels, you can derive your own. You'll see an example of that toward the end of this chapter.

In this chapter, we'll start with the most straightforward panel, the `Canvas`. This panel arranges elements based on X/Y coordinates, with no other factors rolled in. This makes it not only very fast performance-wise but also very easy to learn. From there, we'll take a look at the `StackPanel` and `VirtualizingStackPanel`. These panels provide a little more help layout-wise and are very popular in list controls. Next, we'll explore the most popular panel: the `Grid`. The `Grid` can easily simulate the functions of a `Canvas` or a `StackPanel` plus its own row- and column-based layout functionality. In most applications, the vast majority of nonlist layout work is accomplished via the `Grid` panel.

We'll end this chapter by creating our own `Panel` control. Not only is that useful for advanced applications, but it also provides a lot of practical insight into the layout process covered in chapter 5.

6.1 Canvas

Of all the available panels, the `Canvas` is the simplest and the most straightforward to use. This panel, available since the earliest days of WPF and .NET 3.0,[1] has the least amount of built-in layout logic. Elements on a canvas are positioned simply using left (X) and top (Y) coordinate pairs.

---

[1] `Canvas` was the only panel available in Silverlight 1.0, the version of Silverlight that used JavaScript for all code and layout logic.

The `Canvas` is the right panel to use when you need total control over the layout of elements, without the XAML layout engine attempting to do anything on your behalf. For those reasons, it's both flexible and extremely well performing.

The XAML `Canvas` panel should be not be confused with the HTML5 `canvas` element. The XAML `Canvas` has been around for quite some time and has always been a panel for laying out other elements. The HTML5 `canvas`, in contrast, was designed for pixel-level manipulation and creation of 2D images.

In the coverage of the `Canvas`, I'll start with the basics: how to position child elements using X/Y coordinates. From there, it'll be time to look at the third dimension and consider how controls overlay one another. If you've ever had to implement PowerPoint or CAD-like "Bring to Front" functionality, you'll want to pay attention here. This section wraps up with information on sizing the child elements in the panel.

6.1.1 Positioning in X,Y space

You may have noticed that individual elements in XAML don't have X and Y or Top and Left properties. Instead, positioning is handled with attached properties provided by the `Canvas` type. The use of attached properties (covered in chapter 4) makes it possible to easily support any required layout properties without burdening the control model with a bunch of panel-specific control properties.

Table 6.1 The attached properties provided by the `Canvas`. Use these to position elements within the panel.

Property	Use
`Canvas.Left`	Position the element on the X (horizontal) axis.
`Canvas.Top`	Position the element on the Y (vertical axis).
`Canvas.ZIndex`	Defeat the natural order of elements and position the element in Z space. Elements with a higher `ZIndex` appear layered on top of elements with a lower `ZIndex`. This property is not commonly used. More on this shortly.

Figure 6.2 shows the `Canvas.Left` and `Canvas.Top` properties in action. The first item placed in the panel has a `Canvas.Left` of 2.5 and a `Canvas.Top` of 5. The second has a `Canvas.Left` of 20 and a `Canvas.Top` of 15. The positions are relative to the top left-hand corner of the `Canvas` that contains the elements.

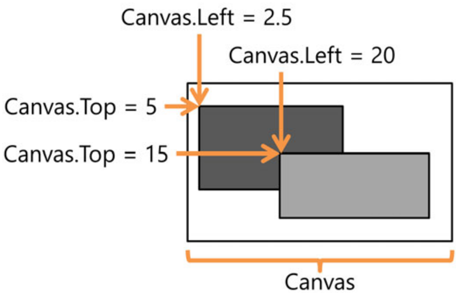


Figure 6.2 Different values for the `Canvas.Left` and `Canvas.Top` properties. Also note that the values are double types and do not need to be whole integers.

ATTACHED PROPERTIES An attached property is a dependency property that's provided by (and stored by) another type, typically a containing panel. Attached properties can be easily identified because they're XAML attributes with dot notation—`Canvas.Top`, for example. For more information, refer to chapter 4.

Of the three positioning properties available with the `Canvas`, the most commonly used properties are `Canvas.Left` and `Canvas.Top`. The following listing shows how to use both of these attached properties. Place this markup inside the default `Grid` tag on the MainPage.xaml of a project created using the Blank App template.

Listing 6.1 `Left` and `Top` attached properties

```
<Canvas Width="300" Height="100" Background="White">
  <TextBlock Canvas.Left="10" Canvas.Top="50"
             Foreground="Black"
             FontSize="20"
             Text="Hello World!" />

  <Rectangle Canvas.Left="60" Canvas.Top="30"
             Width="100" Height="50"
             Fill="Gray" />
</Canvas>
```

Canvas.Top and Canvas.Left

This listing shows two elements, a `TextBlock` and a `Rectangle`, positioned inside a `Canvas`. If you run this sample, you'll see that the gray rectangle overlaps the `TextBlock`, as shown in figure 6.3.

This happens because the `Rectangle` is added to the `Canvas` after the `TextBlock`; therefore it's also rendered after the `TextBlock`. To address this, you need to control the element's Z position.

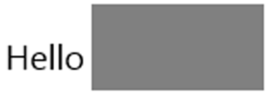


Figure 6.3 Two elements in a white canvas. Note how the rectangle obscures the "World!" part of the "Hello World!" `TextBlock` text.

6.1.2 Controlling the Z position using ZIndex

Imagine a messy stack of papers on your desk. Some papers overlap others, obscuring the details underneath. If you were to recreate the pile by hand, one sheet at a time, you'd first place the bottom-most sheet, then the one on top of that, and the one on top of that, and so on. This is how panels work: The first one becomes the bottom-most element, and each one after that is rendered on top. If they don't overlap at all, you can't see this, but when they do overlap, the order becomes obvious.

The preferred way to address Z ordering at design time is by changing the order in which the elements are added to the panel: Elements are drawn in order so the one lowest in the listing (last one added) is the one drawn on top. It's not always possible to position elements this way, especially if you need to manipulate which ones show up on top at runtime (due to user selection or some other criteria.) In those cases, you can use the `Canvas.ZIndex` property, as shown in the next listing.

Listing 6.2 Positioning elements in Z space using `Canvas.ZIndex`

```
<Canvas Width="300" Height="100" Background="White">
  <TextBlock Canvas.Left="10" Canvas.Top="50"
             Foreground="Black"
             FontSize="20"
             Text="Hello World!"
             Canvas.ZIndex="42"/>

  <Rectangle Canvas.Left="60" Canvas.Top="30"
             Width="100" Height="50"
             Fill="Gray" />
</Canvas>
```

High ZIndex moves element to top of stack

Default ZIndex of 0

As a result of the change in this listing, the full “Hello World!” text is now displayed on top of the rectangle. In this simple example, you could have easily made this change in the markup by moving the elements around. But sometimes elements are added from code, or you need to move them to the top of the display order as a result of a user action. Manipulating the Z index is a good way to accomplish that.

The `Canvas.ZIndex` attached property isn’t an absolute coordinate value like the `Canvas.Top` and `Canvas.Left` coordinate positions. Instead, it represents the relative order of elements *in the same parent panel*. In the previous section, you saw that the rectangle overlapped the text we wanted to show. Manipulating the Z index for either of those two elements will help with arranging them properly.

In the end, the Z index is equivalent to a drawing sort order. The runtime code isn’t necessarily doing a physical rearrangement of elements but rather is providing them to the rendering pipeline in the order of lowest Z index to highest Z index within the scope of the parent panel. Figure 6.4 shows how `Canvas.ZIndex` can be used to position elements.

Items with the same index are rendered based on the order in which they’re added to the panel. For that reason, if you’re going to use Z indexing, it’s generally a good idea to provide each element with a unique number. The numbers don’t have to be sequential, and they can be any `int32` value including negative numbers. The default is zero.

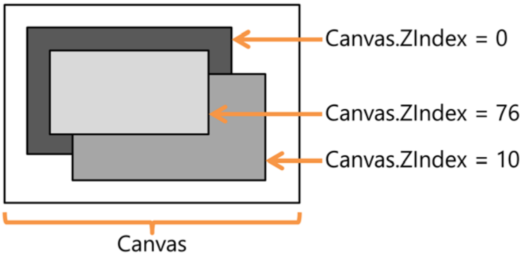


Figure 6.4 **The `Canvas.ZIndex` property is used to control which elements appear on top of others. The higher the Z index, the higher the element is on the layout stack.**

6.1.3 Sizing child elements

Earlier I covered the `HorizontalAlignment`, `VerticalAlignment`, and `Margin` properties. Because of the lack of layout logic beyond X and Y positioning, in the `Canvas`, these have no effect. It's important to note that each element added to a `Canvas` must be given an explicit size or must be a type that provides its own default size during the layout process (such as a `TextBlock`). You'll see later in this chapter that elements in a `Grid` and some other panels can be automatically sized during the layout cycle.

Take the next listing, for example. The first `Rectangle` won't be visible, because it has a size of 0 and doesn't request a default size during layout. If you replace `Canvas` with `Grid` (covered later in this chapter), the `Rectangle` will be visible because the `Grid` will provide a size based on the alignment and margin values.

Listing 6.3 The gray rectangle won't show up because it has no size

```
<Canvas Background="White">
  <Rectangle HorizontalAlignment="Stretch"
             VerticalAlignment="Stretch"
             Margin="100"
             Fill="Gray" />

  <Rectangle Width="50"
             Height="50"
             Fill="Red" />
</Canvas>
```

No effect in a Canvas

Explicit size

In this listing, the first element, the gray rectangle, will be invisible. To fix that, simply assign it a `Width` and `Height`.

The `Canvas` is great for straight X/Y layout. Where it really falls down is when you want to use it to display items that must be positioned based on the position and size of other elements. In that case, if you resize one element, you must go through and manually reposition all the other elements. That's just not what the `Canvas` has been designed for. But there is a set of panels that were built with exactly that in mind.

6.2 StackPanel and VirtualizingStackPanel

Lists of items are very common in applications. Consider menus, list boxes, arrays of photo thumbnails, toolbars, and more. It makes sense, then, that there's a family of panels specifically optimized for display lists. The `StackPanel` and `VirtualizingStackPanel` are that family.

In contrast to the `Canvas`, which requires you to provide a `Canvas.Top` and `Canvas.Left` property for each child, the `StackPanel` performs some intelligent layout but doesn't provide any attached properties for that purpose. Instead, each child added to the `StackPanel` is positioned adjacent to the previous child. Whether the element is positioned to the right of the previous elements or below them is controllable through the `Orientation` property of the panel.

In this section, we'll take a look at the `StackPanel`. First, we'll cover the most basic properties, such as how to set the orientation of the stacking. Then we'll look at an

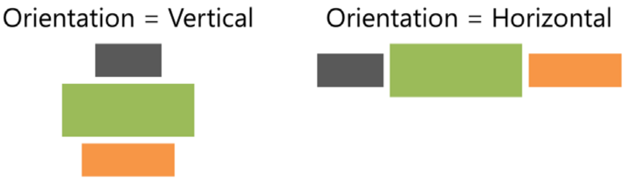


Figure 6.5 Three elements in a `StackPanel` shown in both `Vertical` and `Horizontal` alignments. Notice how the elements are centered by default.

area that often trips up developers and designers new to XAML: sizing children. We'll wrap up this section with information on virtualization. The `VirtualizingStackPanel` is a version of the `StackPanel` control built for smooth performance with large numbers of child elements. In all other ways, it's identical to the `StackPanel`.

6.2.1 Setting the orientation

The primary mechanism for controlling the layout of the `StackPanel` is the `Orientation` property. Setting the property to `Vertical` (the default if no value is provided) arranges children in a vertical column. As you'd expect, setting the property to `Horizontal` arranges the elements in a row. Figure 6.5 shows how elements are laid out when using this property.

The following listing shows how to use the `Orientation` property to align elements vertically as shown in figure 6.5. This approach is often used for traditional lists and vertical menus.

Listing 6.4 `StackPanel` with vertically arranged child elements

```
<StackPanel Orientation="Vertical" Background="White">
  <Rectangle Width="75" Height="35"
             Margin="3" Fill="DimGray" />
  <Rectangle Width="135" Height="55"
             Margin="3" Fill="YellowGreen" />
  <Rectangle Width="95" Height="35"
             Margin="3" Fill="Orange" />
</StackPanel>
```

Vertical orientation

Taller child element

The default orientation is vertical, so it need not be specified. By changing the `Orientation` property to `Horizontal`, you can cause the `StackPanel` to add items left-to-right instead of top-to-bottom.

```
<StackPanel Orientation="Horizontal" ...>
```

The horizontal orientation is useful for left-to-right scrolling lists (think flicking through photos), traditional menus, buttons arranged on an AppBar, and more.

Elements are arranged in the `StackPanel` in the order in which they're added. Unlike the `ZIndex` property in `Canvas`, there's no easy way to rearrange the order of elements once they're in the panel.

Unlike the `Canvas`, the `StackPanel` does provide you with a little assistance in sizing the children.

6.2.2 Sizing children

Unless given an explicit size, the `StackPanel` will be sized to the widest element when in vertical orientation or the tallest element when in horizontal orientation. In contrast to the `Grid`, which we'll cover in the next section, elements added to a `StackPanel` aren't sized in both axes by the panel.

When using the `StackPanel` in vertical orientation, each child element must have an implicit height (due to elements inside the control's template), or you must provide an explicit height. But you can use a `HorizontalAlignment` set to `Stretch` to let the element take up the full width.

Similarly, when the `StackPanel` is in horizontal orientation, each child element must have an implicit or explicit width. You can use the `VerticalAlignment` set to `Stretch` to size the element to the full available height.

Also, because elements are positioned next to each other (like in a `ListBox`), it's important to set margins for items placed in a `StackPanel`.

6.2.3 Virtualizing for performance

When you consider controls such as the `ListBox`, there's the potential for a large number of items to be loaded into the panel. XAML is a retained mode system, so every element loaded incurs a rendering and layout cost. The trick is to reduce the number of items in the visual tree using a technique called *UI virtualization.*

In XAML we have the `VirtualizingStackPanel`—the virtualized counterpart to the `StackPanel`. The `VirtualizingStackPanel` derives from `Panel` (through a few other intermediate classes) just like every other panel covered in this chapter. It doesn't derive from `StackPanel`, however; it is a completely separate implementation that offers an API compatible with `StackPanel`. This is only important if you plan to offer polymorphic methods that are geared to work with a `StackPanel`—consider making them work with the virtualized version as well.

The `VirtualizingStackPanel` can only be used inside of an `ItemsControl` such as a `ListBox`. The `WrapGrid` and `VariableSizedWrapGrid` are similarly restricted panels. Because of this, I'll only briefly cover the `VirtualizingStackPanel` in this chapter, and I'll cover the `WrapGrid` and related panels when I cover the new `ListView` and `GridView` controls in chapter 10.

ENABLING VIRTUALIZATION

If you look at how I added items to the `StackPanel` earlier, you may wonder how on earth you'd write code (or markup) to deal with the virtualization. An important difference between the `VirtualizingStackPanel` and the regular `StackPanel` is that you don't.

This makes complete sense, because virtualization requires the concept of a viewport or viewable subset of items, something that makes sense only if you're in some sort of scrolling container. That means you'll rarely run into a control like this outside of a control template. Nevertheless, to enable virtualization, you can simply use the attached `VirtualizationMode` on the `ItemsControl`, like this:

```
<ListBox VirtualizingStackPanel.VirtualizationMode="Standard"/>
```

The Standard mode creates and discards the item containers, so it's potentially more foolproof, but it isn't the best-performing approach. If you can guarantee that all your items are the same size, the best approach is to use the Recycling mode, as shown here:

```
<ListBox VirtualizingStackPanel.VirtualizationMode="Recycling"/>
```

Recycling the containers avoids creating and destroying elements, which means the costly extra work on the part of the layout engine and memory management are both avoided. With these options, it can be hard to tell if a particular item is currently virtualized, something you may need to concern yourself with if you're doing any custom event work.

CHECKING TO SEE IF AN ELEMENT IS CURRENTLY VIRTUALIZED

The `VirtualizingStackPanel` provides another attached property, `IsVirtualizing`, which may be used on any child element of the panel. This is a bit of an oddball property, though, in that it doesn't behave like a regular attached property. That is, you can't use it from XAML, and you can't set its value. Instead, you'd use it from code via the `GetIsVirtualizing` method of the `VirtualizingStackPanel`, for example:

```
var isVirtual = thePanel.GetIsVirtualizing(childElement);
```

In this case, `thePanel` is the `VirtualizingStackPanel` and `childElement` is the item in the panel that you want to check on.

The two stack panels are appropriate for cases when you want the panel to be sized to the content and to lay out elements in a single dimension. But often you need more flexibility than that. You may want to have elements that take up all available remaining space on a screen, or you may want to lay out elements in more than one dimension. For those tasks, you need the `Grid`.

6.3 Grid

The `Grid` is one of the most useful and flexible of all the panels. Because of that, I use it more than any other panel. It's the default root panel in every XAML template in Visual Studio, and it should also be the first panel you look to when evaluating your layout options.

The `Grid` uses a row and column metaphor for layout. Elements are positioned in a specific row and column and may span as many of each as they wish. In addition, margins and alignment all play key roles in how items are positioned in a `Grid`. The sizes of the rows and columns can be set in specific pixels, autosized to the contents, set to take up the available space, or even set to take up a specific proportion of the available space.

Use the `Grid` when you want to have child elements automatically position and resize based on the available screen real estate. You can have a combination of fixed elements with fixed size and/or position and elements that change size and/or position based on the available space. The `Grid` is key to making your UI work well with different screen resolutions, orientations, and views.

In this section, we'll dive into the `Grid`[2] starting with defining rows and columns. You'll see the different ways the `Grid` supports sizing the boundaries that will hold elements. After that, we'll look at the process for adding items to the grid and how to specify which cell or cells they will occupy. Finally, we'll use alignment and margin properties to modify layout and element size.

6.3.1 Defining rows and columns

Unlike HTML tables, or even similarly named controls like the `DataGrid` we've all used in other platforms, the `Grid` doesn't have the concept of individual cells. Instead, you define rows and columns, and the logical cells are a natural outcome.

The next listing shows how to define rows and columns from XAML.

Listing 6.5 Defining rows and columns from markup

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="42" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="2*" />
    <RowDefinition Height="0.5*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="300" />
  </Grid.ColumnDefinitions>
</Grid>
```

Auto sizing

Star sizing

Pixel sizing

This listing shows the different ways of sizing rows and columns. If you place this markup onto a page and then run it, you'll get a `Grid` with the dimensions shown in figure 6.6. Actually, I take that back. You won't get exactly the same result.

The reason your results will differ is because the markup in listing 6.5 has no content in the `Grid`—there are no child elements. It's important because the `Grid` doesn't have any visible representation of rows and columns; you need to add child elements

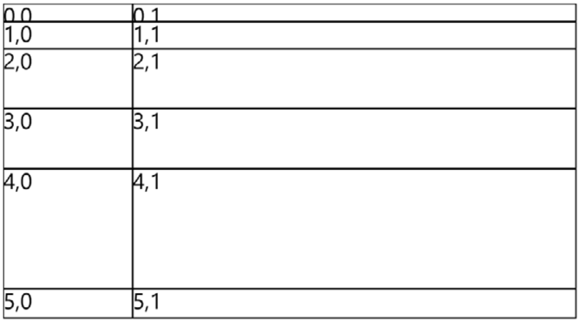


Figure 6.6 The `Grid` created from the markup in listing 6.5. I added rectangles to display borders and `TextBlock` elements to show the `Row,Column` values for each logical cell. Note the clipped content in the top row.

---

[2] Just like Kevin Flynn, but without the cool light cycles.

to see anything. I added a bunch of rectangles with black borders, as well as some `TextBlock` elements with row and column numbers as their text. Whether or not you have elements is even more important with autosizing, such as in the second row down. More on that shortly.

Take a look at the first row in the depicted `Grid`; it was given a pixel size, but the content is too large to be displayed. In the `Grid`, the content is clipped, a feature none of the previous panels (`Canvas and StackPanel`) shared.

Although I used more options with the row definitions, each of the approaches shown in listing 6.5 and table 6.2 works equally with rows and columns.

Table 6.2 The different types of column and row sizing in the `Grid`

Size	Description
Double value	The row or column will be sized in logical pixels given the number provided. If you need an exact size and want all children in the row/column sized to it, this is the approach to use. Examples: “0.5”, “250”
Auto	The row or column will be sized to the content. If there’s no content, the row or column will have a size of zero. Example: “Auto”
Star sizing	The row or column will be sized to the specified number of logical pixels. Add a multiplier to the * to allocate remaining space in something other than even divisions. Examples include “*”, “2*”, “0.5*”

Like most everything else in XAML, you can accomplish in code what you can in markup. Our next listing shows the code equivalent of the listing 6.5 markup, assuming you’ve given the root `Grid` the name `LayoutRoot`.

Listing 6.6 Defining rows and columns from code

```
private void CreateGrid()
{
  Grid g = new Grid();

  for (int i = 0; i < 6; i++)
    g.RowDefinitions.Add(new RowDefinition());

  for (int i = 0; i < 2; i++)
    g.ColumnDefinitions.Add(new ColumnDefinition());

  g.RowDefinitions[0].Height = new GridLength(42);
  g.RowDefinitions[1].Height = new GridLength(1, GridUnitType.Auto);
  g.RowDefinitions[2].Height = new GridLength(1, GridUnitType.Star);
  g.RowDefinitions[3].Height = new GridLength(1, GridUnitType.Star);
  g.RowDefinitions[4].Height = new GridLength(2, GridUnitType.Star);
  g.RowDefinitions[5].Height = new GridLength(0.5, GridUnitType.Star);

  g.ColumnDefinitions[0].Width = new GridLength(1, GridUnitType.Star);
  g.ColumnDefinitions[1].Width = new GridLength(300);

  LayoutRoot.Children.Add(g);
}
```

Create rows

Create columns

Auto sizing

Star sizing

Pixel sizing

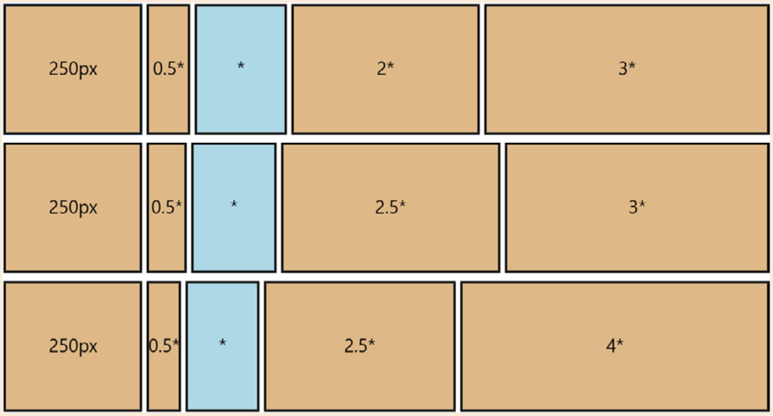
Creating rows and columns in the `Grid` is easy both from markup and from code. I mentioned that we don't have the concept of cells. That's important because all children are children of the grid, not children of cells within the grid as they would be in HTML. You position them by providing the row and column for the element to be positioned in.

More on grid star sizing

Star sizing can be a little confusing. That's because the sizes are based on all the other sizes and not some fixed amount. Simply calculating the initial value is a complex recursive calculation, for example:

- * and 1.0* are equivalent and mean a single unit.
- 3* means three times the amount that would have been allocated to 1.0*.
- 0.5* means one-half the amount that would have been allocated to 1.0*.

But the space that would be allocated to * must be recalculated to take into account everything else.



You can see from this figure that the amount of space allocated to the * sized column changes based on how much space the other columns request. The only value that stays the same is the fixed-pixel layout.

Calculating this recursive layout in your head or on paper has been shown to cause brain-level stack overflow and a deep hatred for flexible and fluid layout. Given that, I suggest experimenting on a blank page before applying a complex star-sizing layout to your own project.

6.3.2 *Adding and positioning elements in rows and columns*

Whether automatic or explicit, one important service all panels provide is positioning child elements. In support of this, the `Grid` provides four attached properties, shown in table 6.3.

Table 6.3 The attached properties provided by the `Grid`. Use these to position elements within the panel.

Property	Use
`Grid.Row`	A zero-based index into the rows defined by the `RowDefinitions` collection.
`Grid.Column`	A zero-based index into the columns defined by the `ColumnDefinitions` collection.
`Grid.ColumnSpan`	The number of columns this element takes up. Default value is 1.
`Grid.RowSpan`	The number of rows this element takes up. Default value is 1.

The default `Grid.Row` and `Grid.Column` are both 0, so they don't need to be specified. I always do, however, because it makes the intent clear.

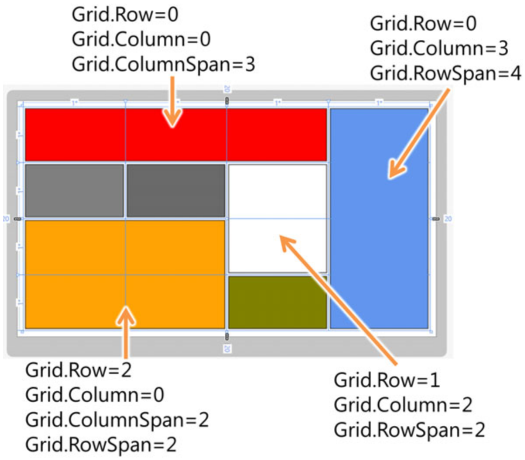

Grid.Row=0
Grid.Column=0
Grid.ColumnSpan=3
Grid.Row=0
Grid.Column=3
Grid.RowSpan=4
Grid.Row=2
Grid.Column=0
Grid.ColumnSpan=2
Grid.RowSpan=2
Grid.Row=1
Grid.Column=2
Grid.RowSpan=2

Figure 6.7 A bunch of rectangles in a grid, viewed in the Visual Studio XAML designer. Any elements with row or column spanning have been called out so you can see the properties.

Figure 6.7 shows a `Grid`, in the Visual Studio designer, with seven child rectangles. Two of the rectangles span columns, three span rows, and three simply take up their own cell. All of the rectangles have margins, so you can see how the margins come into play with layout, especially with row and column spanning.

The next listing shows how to use these four properties to create a grid with flexible positioning of elements. The created grid is the one illustrated in figure 6.7. Note how the order in which elements are added doesn't matter as long as they don't overlap. Every element is added inside the `Grid` tag, after the row and column definitions.

Listing 6.7 Positioning child elements in the grid

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
```

Four even rows

Four even columns

```
  </Grid.ColumnDefinitions>

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="Red" Margin="5"
             Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3" />

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="Gray" Margin="5"
             Grid.Row="1" Grid.Column="0" />
  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="DimGray" Margin="5"
             Grid.Row="1" Grid.Column="1" />
  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="White" Margin="5"
             Grid.Row="1" Grid.Column="2" Grid.RowSpan="2"/>

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="CornflowerBlue" Margin="5"
             Grid.Row="0" Grid.Column="3" Grid.RowSpan="4" />

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="Orange" Margin="5"
             Grid.Row="2" Grid.Column="0"
             Grid.RowSpan="2" Grid.ColumnSpan="2" />

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="Olive" Margin="5"
             Grid.Row="3" Grid.Column="2" />
</Grid>
```

Span three columns

Span two rows

Span four rows

Span two rows and two columns

A `Grid` with no rows or columns defined is treated as though it has a single row and column. In addition, if you specify a row or column outside the range created by the row and column definitions, the element will be positioned at the closest defined row or column. So, if there are six columns (indexed 0–5) and you specify a `Grid.Column` value of 10, it will be treated as an index of 5. If you don't specify a row or column, the default value is 0.

Beyond simply positioning elements, the `Grid` provides a lot of help with sizing elements.

6.3.3 Using alignment and margins for sizing and positioning

Look back at figure 6.7. Notice how the rectangles take up all available space allotted to them. Now look at listing 6.7. Unlike the `Canvas` example, the rectangles here have no defined `Width` or `Height`. Instead, when elements are placed in a `Grid`, they are allowed, by default, to stretch to the full height and width available in the logical cell.

In addition, if the elements have sizes, you can use the `HorizontalAlignment` and `VerticalAlignment` properties to position them on the screen, for example, in a corner. This can be useful when you want to have an element overlay other elements on the grid. You can use these alignment properties together with margins and row and

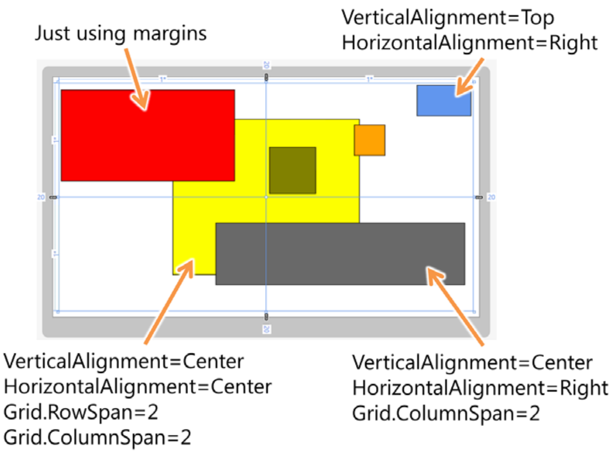

Just using margins
VerticalAlignment=Top
HorizontalAlignment=Right
VerticalAlignment=Center
HorizontalAlignment=Center
Grid.RowSpan=2
Grid.ColumnSpan=2
VerticalAlignment=Center
HorizontalAlignment=Right
Grid.ColumnSpan=2

Figure 6.8 Effective use of alignment and margins to align elements in different parts of the grid

column spanning to be able to position any element in any spot in the grid, as shown in figure 6.8.

The following listing shows how to position elements in a grid to create the layout shown in figure 6.8. This same technique works within single cells and in spanned cells.

Listing 6.8 Positioning and sizing elements using alignment and margins

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="Yellow" Margin="25"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"
             Height="500" Width="600"
             Grid.Row="0" Grid.Column="0"
             Grid.ColumnSpan="2" Grid.RowSpan="2"/>

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="Red" Margin="5,20,100,50"
             Grid.Row="0" Grid.Column="0" />

  <Rectangle Stroke="Black" StrokeThickness="2"
             Fill="CornflowerBlue" Margin="5"
             HorizontalAlignment="Right"
```

Two even rows

Two even columns

Background rectangle spans all

Margins for size and position

```
                VerticalAlignment="Top"
                Height="100" Width="175"
                Grid.Row="0" Grid.Column="1" />

     <Rectangle Stroke="Black" StrokeThickness="2"
                Fill="Orange"
                HorizontalAlignment="Center"
                VerticalAlignment="Center"
                Height="100" Width="100"
                Grid.Row="0" Grid.Column="1" />

     <Rectangle Stroke="Black" StrokeThickness="2"
                Fill="Olive" Margin="10"
                HorizontalAlignment="Left"
                VerticalAlignment="Bottom"
                Height="150" Width="150"
                Grid.Row="0" Grid.Column="1" />

     <Rectangle Stroke="Black" StrokeThickness="2"
                Fill="DimGray" Margin="25"
                HorizontalAlignment="Right"
                VerticalAlignment="Center"
                Height="200" Width="800"
                Grid.Row="1" Grid.Column="0"
                Grid.ColumnSpan="2"/>
</Grid>
```

Centered in top right cell

Span bottom cells

This listing shows a number of techniques for positioning elements in a `Grid`. You can see the precedence of sizing: Explicit `Height` and `Width` have higher precedence than margins or a `Stretch` alignment. If aligned to an edge, only the margin for that edge comes into play. If that edge moves, the element will move with it to maintain its distance. This is similar to "docking" elements in Windows forms. If sized using only margins with `Stretch` for `HorizontalAlignment` and `VerticalAlignment`, the element will take up all available space in the cell (or spanned cells).

The `Grid` is easily my favorite panel in XAML. Unless you're working with an `ItemsControl` like a `ListBox`, where elements must be arranged automatically without any attached properties, you can accomplish just about any layout task you need by using it.

Making effective use of margins and alignment instead of hardcoding sizes will let you automatically adjust for different resolutions and layouts. This is one of the most important skills to master to allow your apps to run on the multitude of formats and orientations supported by Windows Modern Style apps.

Explicitly sizing elements and aligning them to one or more edges enables you to "dock" elements like status information, toast-like notifications, user sign-in information, AppBars, and more. It really is that flexible.

We've covered a number of panels so far, each with its own way of laying out the elements provided. What happens, though, when these panels don't do quite what you want? Do you just hack something together using a `Canvas` and a bunch of random code in code-behind? No! You create your own panel; it's easy, as you'll see.

6.4 Creating a custom panel

As I mentioned, the primary responsibility for positioning and sizing elements falls to the panel the elements are contained in. You've seen in this chapter that some panels, such as the `Canvas`, position using simple `Left` and `Top` coordinates. Others, such as the `StackPanel`, lay out children based on a series of measurements and a placement algorithm.

In this section, we're going to build a simplified version of a panel that doesn't currently exist in Windows: the `OrbitPanel`. This panel will lay out elements in a circle rather than using the horizontal and vertical options available with the stock `StackPanel` or the row and column layout of a `Grid`. The new panel in action can be seen in figure 6.9.

The `OrbitPanel` control supports an arbitrary number of orbits, as long as that number is greater than zero. Each orbit is a concentric circle starting at the center point of the panel. The amount of space allocated to an orbit is a function of the number of orbits and the size of the panel itself. If there are many orbits, the space will be narrower.

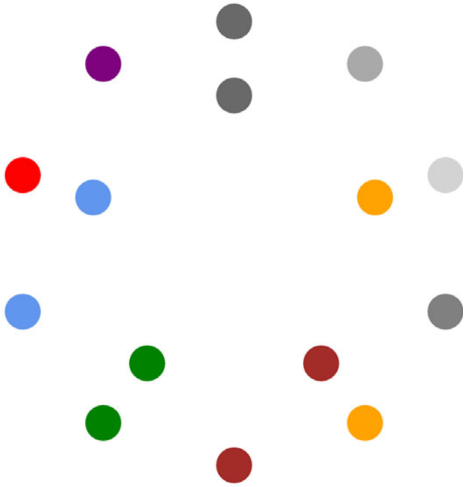


Figure 6.9 The `OrbitPanel` in action. The inner orbit (level 1) has five ellipses. The outer orbit (level 2) has 10 ellipses.

For each orbit, the layout is done by starting at angle 0 and equally dividing the remaining degrees by the number of items in the specific orbit. Items added to the panel may specify an orbit via the use of an attached property. No rotation is performed on the elements, just positioning.

In this section, you'll build this panel, starting with project creation, including the addition of a library project specifically made for this panel. You'll create a dependency property as well as an attached property, both because they're useful and because creating them is a necessary skill for panel and control builders. From there, you'll spend most of the time looking at how to perform the measure and arrange steps described in chapter 6 to lay out the child elements.

6.4.1 Project setup

For this example, create a new Windows Store XAML Blank App project named CustomPanelExample. Once the solution is up, add a Class Library (Windows Store App) project named ControlsLib (right-click the solution, select Add > New Project). Though I could've put the custom panel into the same project as the app, that's almost never the way reusable controls are developed.

From the app project, add a project reference to the ControlsLib project. Do this by right-clicking the app project file, selecting Add Reference, navigating to the Projects tab, and selecting the project. While you're in the ControlsLib project, remove the default class1.cs file that came with the template.

With the project structure in place, let's work on the `OrbitPanel` class.

6.4.2 *The OrbitPanel class*

Inside the ControlsLib project, delete the default class1.cs file and add a new class named `OrbitPanel`. This class will contain all the code for the custom panel. Derive the class from the `Panel` base type, as shown in the next listing. Make sure you mark the class as `public`, or it won't be visible to the test project.

Listing 6.9 The empty `OrbitPanel` class

```
using System;
using System.Collections.Generic;
using System.Linq;
using Windows.Foundation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace ControlsLib
{
  public class OrbitPanel : Panel
  {
  }
}
```

Panel namespace

Make sure it's public!

In addition to the `Children` property covered at the start of this chapter, the `Panel` base class provides a `Background` brush property and a Boolean `IsItemsHost`, which is used in concert with the `ItemsControl` class. Deriving from `Panel` allows you to substitute your panel for the `StackPanel` in a `ListBox`, for example. The `Background` property allows you to provide a background color for the panel.

The `OrbitPanel` class will need to have two properties. The first, `Orbits`, will control the number of concentric circles, or orbits, available for placing items. The second is an attached property, `Orbit`, to be used on items placed into the panel; it controls which circle, or orbit, the item is to be placed in.

6.4.3 *Orbits dependency property*

In general, controls and panels should expose properties as dependency properties. If there's any possibility that they'll be used in binding or animation, a dependency property is the way to go.

Dependency properties are specified at the class level using a static property and `DependencyProperty.Register` call. For use in code and XAML, they're also wrapped with a standard property wrapper that internally uses the dependency property as the backing store. Optionally, the dependency property may specify a callback function to be used when the property changes.

The following listing shows the complete definition for the `Orbits` property, with all three of these items in place.

Listing 6.10 The `Orbits` property

```
public int Orbits                                         <─┐ Wrapper property
{
  get { return (int)GetValue(OrbitsProperty); }
  set { SetValue(OrbitsProperty, value); }
}
public static readonly DependencyProperty OrbitsProperty =
  DependencyProperty.Register("Orbits",
                              typeof(int),
                              typeof(OrbitPanel),
                              new PropertyMetadata(1, OnOrbitsChanged));
private static void OnOrbitsChanged(DependencyObject d,
                                    DependencyPropertyChangedEventArgs e)
{
  if ((int)e.NewValue < 1)
  {
    throw new ArgumentException(
                    "Orbits must be greater than or equal to 1.");
  }
}
```

The first thing you'll notice in this code is the `Orbits` property. This is a standard property wrapper, used for simple property access in code and required for property access in XAML. The property code uses the `GetValue` and `SetValue` methods, provided by `DependencyObject`, to access the backing dependency property. Though not required at a compiler or framework level (unless you want to use the property in XAML), providing the wrapper is a standard practice when defining dependency properties.

> **TIP** Visual Studio 2012 includes a snippet for declaring dependency properties. An easy way to get to them is to right-click in the code editor and select Insert Snippet. From there, look under the NETFX30 option, because this snippet was first introduced with WPF. This snippet also works well for Silverlight applications and saves you from remembering the exact syntax.

The next chunk of code in this listing defines and registers the dependency property. The single line both defines the property and registers it with the property system. The first parameter is the string name of the property. By convention, the name of the dependency property variable is this string plus the word *Property*. The second parameter is the type of the property itself—in this case, an `int`. The third parameter is the type you're registering the property on. The fourth and final parameter is a `PropertyMetadata` object.

The `PropertyMetadata` object can be used to specify a default value, a property changed callback, or, as seen here, both. When providing the default property value, be specific with the type. For example, a property value of `1` won't work with a double type; you must specify `1.0` or face the wrath of an obscure runtime error.

The property changed callback function enables you to hook into the process to perform actions when the dependency property changes. Note that you'd never want to do this inside the property wrapper, because that would only catch a few of the scenarios under which the property could change. The callback function takes in an instance of the object that owns the property, as well as an event args class that has both the new and old values available for inspection.

All three pieces—the property wrapper, the dependency property definition and registration, and the callback function—make up the implementation of a single dependency property in Windows. Though verbose, the benefits provided by dependency properties are great, as shown throughout this book. When creating your own properties for panels and controls, err on the side of implementing them as dependency properties.

A specialized type of `DependencyProperty`, the attached property is used when you want to provide a way to enhance the properties of another object. That's exactly what you need to do with the `Orbit` property.

6.4.4 Orbit attached property

Each item added to the `OrbitPanel` needs to be assigned to a specific circle or orbit. This is similar in concept to how a `Grid` needs items to specify rows and columns or how the `Canvas` needs `Left` and `Top` for each element. The association between the orbit number and the item's position is shown in figure 6.10.

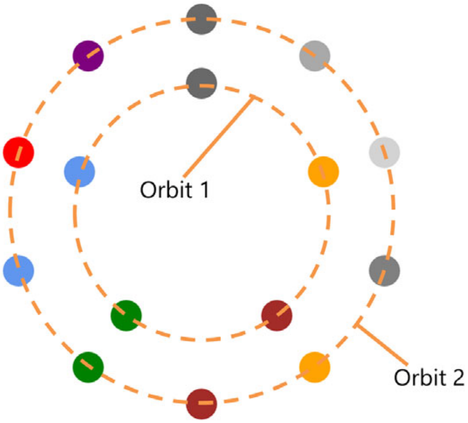


Figure 6.10 Orbits are numbered starting with zero: the innermost orbit. Orbit zero isn't used in this example because that would be too close to the center for good layout.

You'll do the same with the `Orbit` property of the `OrbitPanel`. The next listing shows the implementation of the `Orbit` attached property in the `OrbitPanel` class. Add this into the class now.

Listing 6.11 The `Orbit` attached property in the `OrbitPanel` class

```
public static int GetOrbit(DependencyObject obj)
{
  return (int)obj.GetValue(OrbitProperty);
}
public static void SetOrbit(DependencyObject obj, int value)
```

```
{
  obj.SetValue(OrbitProperty, value);
}
public static readonly DependencyProperty OrbitProperty =
    DependencyProperty.RegisterAttached("Orbit",
                                    typeof(int),
                                    typeof(OrbitPanel),
                                    new PropertyMetadata(0));
```

Note that attached properties don't use a wrapper. Instead, following convention, you provide `Get` and `Set` methods to allow the properties to be used in code and XAML.

The `RegisterAttached` method is similar to the `Register` method shown in listing 6.10, with the parameters being identical. In this case, you don't use a callback method but instead provide a default value of zero.

The public-facing API of the `OrbitPanel` is now complete. The next listing is the MainPage.xaml markup to use and populate the `OrbitPanel`. Note the `controls` namespace declaration at the top.

Listing 6.12 Controls arranged inside the `OrbitPanel`

```
<Page
  x:Class="CustomPanelExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:CustomPanelExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:controls="using:ControlsLib"
  mc:Ignorable="d">

  <Grid Background="White">
    <controls:OrbitPanel Orbits="3" Margin="75">
      <Ellipse controls:OrbitPanel.Orbit="1"
               Width="50" Height="50" Fill="DimGray"/>
      <Ellipse controls:OrbitPanel.Orbit="1"
               Width="50" Height="50" Fill="Orange"/>
      <Ellipse controls:OrbitPanel.Orbit="1"
               Width="50" Height="50" Fill="Brown"/>
      <Ellipse controls:OrbitPanel.Orbit="1"
               Width="50" Height="50" Fill="Green"/>
      <Ellipse controls:OrbitPanel.Orbit="1"
               Width="50" Height="50" Fill="CornflowerBlue"/>

      <Ellipse controls:OrbitPanel.Orbit="2"
               Width="50" Height="50" Fill="DimGray"/>
      <Ellipse controls:OrbitPanel.Orbit="2"
               Width="50" Height="50" Fill="DarkGray"/>
      <Ellipse controls:OrbitPanel.Orbit="2"
               Width="50" Height="50" Fill="LightGray"/>
      <Ellipse controls:OrbitPanel.Orbit="2"
               Width="50" Height="50" Fill="Gray"/>
```

OrbitPanel namespace

OrbitPanel

Orbit attached property

```
        <Ellipse controls:OrbitPanel.Orbit="2"
                 Width="50" Height="50" Fill="Orange"/>
        <Ellipse controls:OrbitPanel.Orbit="2"
                 Width="50" Height="50" Fill="Brown"/>
        <Ellipse controls:OrbitPanel.Orbit="2"
                 Width="50" Height="50" Fill="Green"/>
        <Ellipse controls:OrbitPanel.Orbit="2"
                 Width="50" Height="50" Fill="CornflowerBlue"/>
        <Ellipse controls:OrbitPanel.Orbit="2"
                 Width="50" Height="50" Fill="Red"/>
        <Ellipse controls:OrbitPanel.Orbit="2"
                 Width="50" Height="50" Fill="Purple"/>
      </controls:OrbitPanel>
    </Grid>
</Page>
```

This example shows how you can add elements to the `OrbitPanel` and also set the custom attached properties. You haven't implemented any layout, so you won't see anything just yet.

Dependency properties—and the special type of dependency property, the attached property—are essential and often use parts of the property system in Windows XAML. When creating your own panels and controls, you'll almost certainly rely on them as the primary means of providing "knobs" your users can use to control the behavior of your custom classes.

In the case of the `OrbitPanel`, both of these properties will come into play when performing a custom layout.

6.4.5 Custom layout

The main responsibility of a panel is the layout of its child controls. This is primarily what makes an element a panel. A panel that performed no custom layout wouldn't be particularly useful. In our panel, each child element will be arranged starting at the top center and proceeding clockwise on an even angle increment. Figure 6.11 shows the layout for each orbit.

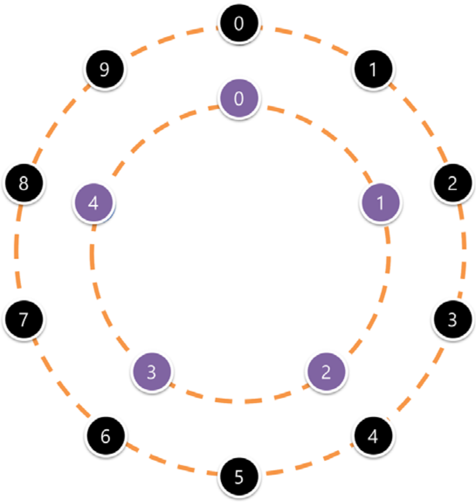


Figure 6.11 Each orbit has its elements positioned starting clockwise from 12:00 (zero degrees).

As you learned in chapter 5, the layout pass involves two steps: measure and arrange. The measure step measures all the children of the panel, as well as the overall panel itself. The arrange step performs final placement of the children and sizing of the panel. As the authors of a custom panel, it's our responsibility to provide this critical functionality. The following listing shows the measure step, implemented in the `MeasureOverride` method of the `OrbitPanel` class.

Listing 6.13 The measure step

```
protected override Size MeasureOverride(Size availableSize)
{
  var sortedItems = SortElements();
  double max = 0.0;
  foreach (List<UIElement> orbitItems in sortedItems)
  {
    if (orbitItems.Count > 0)
    {
      foreach (UIElement element in orbitItems)
      {
        element.Measure(availableSize);

        if (element.DesiredSize.Width > max)
          max = element.DesiredSize.Width;
        if (element.DesiredSize.Height > max)
          max = element.DesiredSize.Height;
      }
    }
  }
  Size desiredSize = new Size(max * Orbits * 2, max * Orbits * 2);
  if (double.IsInfinity(availableSize.Height) ||
      double.IsInfinity(availableSize.Width))
        return desiredSize;
    else
        return availableSize;

}
```

Measure each child

Return panel measurements

The measure pass starts by getting a list of all items, grouped by their orbit using the `SortElements` function defined in the next listing. The code then loops through each orbit, then through each item in the orbit, and measures that item. It then gets the largest dimension (either width or height) from that element and compares it to the current max. This is admittedly a bit of a hack, because the size allotted to each item is, in theory, a pie slice, not a rectangle. In addition, because of the simplified nature of the orbit sizing, you don't need to group the children by orbit. Nevertheless, it'll work for this example.

Once the code has looped through every child item, it then calculates the desired size for this panel. That's calculated by taking the number of orbits, multiplying by two to account for the circular nature, then multiplying by the maximum item size. If the original size passed in was unlimited, the code returns the desired size; otherwise, it returns the size provided to the control.

The next listing relies on the `SortElements` function. The code for both that function and the `CalculateOrbitSpacing` function (used in the arrange step) is included in the next listing.

Listing 6.14 Supporting functions

```
private double CalculateOrbitSpacing(Size availableSize)
{
  double constrainingSize = Math.Min(
              availableSize.Width, availableSize.Height);
  double space = constrainingSize / 2;
  return space / Orbits;
}
private List<UIElement>[] SortElements()
{
  var list = new List<UIElement>[Orbits];
  for (int i = 0; i < Orbits; i++)
  {
    if (i == Orbits - 1)
      list[i] = (from UIElement child in Children
                 where GetOrbit(child) >= i
                 select child).ToList<UIElement>();
    else
      list[i] = (from UIElement child in Children
                 where GetOrbit(child) == i
                 select child).ToList<UIElement>();
  }
  return list;
}
```

`CalculateOrbitSpacing` uses the size of the panel to figure out the spacing of the individual concentric circles. This is done by evenly dividing the total space. The `SortElements` function takes each child and puts it into a list by orbit, just to make it easier to process.

Note that the `SortElements` function has special logic to group any elements in an invalid orbit into the highest orbit, much like the `Grid` does with invalid row and column values. It doesn't handle any error cases where a negative orbit number was specified, but that's easy enough to add.

The most important step in the `MeasureOverride` code is the step that measures each child. That's what sets the desired size for each child in preparation for the arrange step shown next, wrapping up the code for the `OrbitPanel` class.

Listing 6.15 The arrange step

```
protected override Size ArrangeOverride(Size finalSize)
{
  var sortedItems = SortElements();
  double orbitSpacing = CalculateOrbitSpacing(finalSize);

  int i = 0;
  foreach (List<UIElement> orbitItems in sortedItems)
  {
    int count = orbitItems.Count;
    if (count > 0)
    {
```

Place child in final location

```
      double circumference = 2 * Math.PI * orbitSpacing * (i + 1);
      double slotSize = Math.Min(orbitSpacing, circumference / count);
      double maxSize = Math.Min(orbitSpacing, slotSize);
      double angleIncrement = 360 / count;
      double currentAngle = 0;
      Point centerPoint =
                 new Point(finalSize.Width / 2, finalSize.Height / 2);
      foreach (UIElement element in orbitItems)
      {
        double angle = Math.PI / 180 * (currentAngle - 90);
        double left = orbitSpacing * (i + 1) * Math.Cos(angle);
        double top = orbitSpacing * (i + 1) * Math.Sin(angle);
        Rect finalRect = new Rect(
              centerPoint.X + left - element.DesiredSize.Width / 2,
              centerPoint.Y + top - element.DesiredSize.Height / 2,
              element.DesiredSize.Width,
              element.DesiredSize.Height);
        element.Arrange(finalRect);
        currentAngle += angleIncrement;
      }
    }
    i++;
  }
  return base.ArrangeOverride(finalSize);
}
```

The arrange step is where the real layout happens. It's in this function that the individual children are placed in their final locations. This is the function that requires digging way back to 10th or 11th grade to remember that trigonometry.

This function, like the previous one, starts by sorting the children into their respective orbits. Again, this is performed by the `SortElements` function. It then runs through each orbit, calculating the size of the circle and the angular offset of each item. The angle chosen is based on the number of items in that orbit; it's 360 degrees evenly divided by the item count.

Then, the function calculates the left and top position given the angle. This left and top will actually be used for the center point of the element being placed. This can lead to some elements being cut off, which is why I provided a margin in the markup that uses this panel. With that all calculated, this function calls `Arrange` on the element to move it to its final location.

If you run the application now, you'll see the image from the opening of this section (figure 7.9), with two orbits of buttons. There's a lot you could do to enhance a panel like this, including enclosing each element in a host container that you rotate to match the angle. You could even write a panel that lays out elements in a spiral fashion, maybe call it `TornadoPanel` or `DrainPanel`; the latter of course would need to take into account the hemisphere of the user to decide on a clockwise or counterclockwise rotation.[3]

---

[3] Yes, I know that's not true, but I never let facts get in the way of an amusing story. No, the Coriolis effect doesn't impact how sinks or tubs drain. http://en.wikipedia.org/wiki/Coriolis_effect.

Panels are all about measuring and arranging their children. Measuring is used to ask each child what size it wants to be and to provide the overall size for the panel. Arranging is used to calculate the final location of each of the child elements.

6.5 Summary

Panels are one of the essential topics to learn when getting into WinRT XAML. If you're a Silverlight, Windows Phone, or WPF developer, you'll have a huge head start because the layout system and panel infrastructure are almost identical. The `Canvas`, `StackPanel`, and `Grid` are virtually unchanged from all other versions of XAML.

The `Canvas` is the best-performing panel because it imposes almost no layout overhead. Instead, it positions elements based on provided `Canvas.Left` and `Canvas.Top` (X and Y) coordinates. Use it when you need something lightweight and flexible.

The `StackPanel` and `VirtualizingStackPanel` provide a little more layout smarts. They will automatically arrange items in the direction specified by the `Orientation` property. Use them when you need to display items in a horizontal or vertical list.

The most popular panel is the `Grid`. This should be your go-to panel for most layout tasks. You can arrange items in a grid/cell layout by rows and columns, you can span rows or columns with elements, and you can position elements using margins and alignment. It's one of the heaviest grids, so it's not necessarily the one to pick if you are down to critical levels of performance, but it will save you a ton of time when designing the UI.

Finally, you have your own panel. Once you've exhausted everything the built-in panels can do, you can create your own panel. In this case, it's a panel that arranged items using circular orbits around a central point.

This chapter completes our discussion of layout. In the next chapter, we'll look at brushes and graphics and how to use resources and styles.

7

Brushes, graphics, styles, and resources

This chapter covers

- Brushes and colors
- Resources and styles
- Vector graphics
- Images

The Windows 8 design aesthetic generally frowns on decoration and ornamentation (also referred to as “chrome”). That doesn’t mean that graphics have no role to play. In comparison to other versions of Windows, Windows 8 is far more graphically rich with a stronger focus on colorful touch-friendly shapes and vibrant imagery. Just take a look at the apps from the Windows Store or the apps that come preinstalled—you’ll find that they make extensive use of both bitmap and vector graphics.

When I first started working with Silverlight 1.0, we had the `Canvas` for the panel and some basic vector and bitmapped shapes for graphics. We had to build everything else on top of that. I remember I built buttons and combo boxes using only these most primitive building blocks, while walking uphill, barefoot in the snow, both ways!

Happily, we don’t need to re-create the wheel here; the controls have been created for us. But when you look at the controls and their templates in the next chapters, you’ll see they’re made up of panels, shapes, and other controls, which are ultimately made up of other panels and shapes. You'll use the brushes to paint those controls as well as the text that's so prevalent in Windows 8 apps.

In this chapter, I'll start with coverage of the brushes, focusing on how colors are defined and used and how brushes make it possible to have solid colors as well as gradients. I'll even get into coverage of the `ImageBrush`, which enables you to paint shapes and even text using a bitmapped image.

Then, because brushes and colors are commonly stored as resources in the application or in standard resource dictionary files, we'll take a little detour into resource management. This is an important topic for this chapter and is easy to grasp with the simple resources I'll introduce. Following right on the heels of the discussion of resources, we'll implement a very common type of resource: a style. Styles in XAML are an important way to standardize not only visible properties (as in CSS) but even content and other properties. They also tend to make extensive use of resources, which makes them a natural follow-on to that discussion.

Next, it'll be time to look into vector graphics. We'll take a look at lines, polylines (and polygons), ellipses, rectangles, and even the `Path` mini language for defining complex shapes. From there, we'll wrap up the chapter with a look at using bitmapped images outside the context of the `ImageBrush` covered earlier. Together, brushes, bitmapped graphics, and vector graphics represent the heart of the 2D drawing system in WinRT. Resources make working with them more pleasant.

7.1 Brushes

In some platforms there's a distinction made between a pen and a brush: Pens are for drawing lines and brushes are for filling in shapes. In XAML, there's no such thing as a pen: All painting, whether it be lines, gradients, fills, or backgrounds, is accomplished using a brush.

Just like in the real world, a brush is a device used to paint a color on something. Unlike the real world, you don't have to spend 20 minutes cleaning it in-between uses. Also, in XAML, the colors can be sold colors, smooth transitions between colors, or even an image. The idea of a brush is itself an abstract concept. It's the concrete specializations that we'll use in XAML.

The Windows 8 design aesthetic encourages the use of a palette of solid colors, so we'll start with the `SolidColorBrush`. From there, we'll look at the `LinearGradientBrush`. This brush is good for adding subtle color variation to your UI. We'll wrap up with the `ImageBrush`—a way to paint shapes and even text using bitmapped images. In all cases, we'll cover the XAML approach as well as the C# approach for working with the brushes.

7.1.1 Solid-color brushes

The simplest brush is the `SolidColorBrush`. As the name implies, this is a brush that's made up of a single solid color, without any variation. Whenever you specify a color code or named color into a property in XAML, you're using a `SolidColorBrush`. A type converter (code invoked whenever converting, in this case, a string to a type) converts the color's string representation into the solid-color brush. In this listing all four rectangles will be the same color.

Listing 7.1 Four ways of specifying a red `SolidColorBrush`

```
<StackPanel>
  <Rectangle Fill="Red" Stroke="White"
             Width="100" Height="100" />
  <Rectangle Fill="#FFFF0000" Stroke="#FFFFFFFF"
             Width="100" Height="100" />
  <Rectangle Stroke="#FFFFFFFF"
             Width="100" Height="100">
    <Rectangle.Fill>
      <SolidColorBrush Color="Red" />
    </Rectangle.Fill>
  </Rectangle>
  <Rectangle Stroke="#FFFFFFFF"
             Width="100" Height="100">
    <Rectangle.Fill>
      <SolidColorBrush>
        <SolidColorBrush.Color>
          Red
        </SolidColorBrush.Color>
      </SolidColorBrush>
    </Rectangle.Fill>
  </Rectangle>
</StackPanel>
```

Color names

Color hex codes

Red brush explicit

Red brush expanded

In this listing, you can see four different variations on specifying the `Fill` brush for a `Rectangle`. The first uses properties with named colors. The second uses the hex color code for red. The third breaks out the brush using property element syntax (you can actually leave out the `SolidColorBrush` and just use `Red` inline, as shown in chapter 3 on XAML). The fourth is extra verbose, completely breaking out the full-brush specification.

I show all methods here because you're going to use variations on these for other brushes. It's important to understand that these are all simply different ways of specifying the same result: a solid `Fill` color of `Red`.

When it comes to the colors themselves, you can see that I specified them in two different ways: color names and color codes. The color codes are like those you'd use in HTML, with an alpha component. The format is #AARRGGBB, which, in addition to looking like an expletive from a pirate with a swollen lip, translates to 1 byte each of alpha, red, green, and blue. Full opaque alpha is 0xFF (255) for the first byte; fully transparent is 0x00 (0). Figure 7.1 shows the breakdown of the color code.

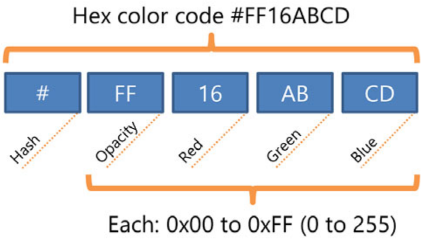


Figure 7.1 The components of a hex color code

Unlike with Silverlight, the full set of color names is available in both code and XAML. You've seen the color names in use in XAML already. In code, the `Windows.UI.Colors` class provides the list of color names. But unlike the case with WPF, you don't have access to the system theme colors. Rather than attempt to match the system colors, you should use your own colors and branding consistently within your app.

The next listing shows several different ways to create a `SolidColorBrush` from code.

Listing 7.2 `SolidColorBrush` in code

```
public MainPage()
{
  this.InitializeComponent();
  SetColors();
}

private void SetColors()
{
  LayoutRoot.Background = new SolidColorBrush(Colors.Red);

  var color = Color.FromArgb(0xFF, 0xFF, 0x00, 0x00);
  LayoutRoot.Background = new SolidColorBrush(color);

  var color2 = ColorHelper.FromArgb(0xFF, 0xFF, 0x00, 0x00);
  LayoutRoot.Background = new SolidColorBrush(color2);

  var brush = new SolidColorBrush();
  brush.Color = color;
  LayoutRoot.Background = brush;

  string xmlns =
    "http://schemas.microsoft.com/winfx/2006/xaml/presentation";
  string xaml =
    "<SolidColorBrush xmlns=\""+ xmlns +"\" Color=\"#FFFF0000\" />";
  var brush2 = (Brush)XamlReader.Load(xaml);
  LayoutRoot.Background = brush2;
}
```

Use constructor

Use Color

Use ColorHelper

Use color property

Parse XAML

As with most of my examples, `LayoutRoot` is the name I gave to the root `Grid` on the page. The first example uses the constructor of `SolidColorBrush` to specify the color using a color name. The second uses the `Color.FromArgb` method with the individual color components. The third does the same thing but with the `ColorHelper` class. As you can see, the `ColorHelper.FromArgb` method works in C#, but it's really meant just for JavaScript apps. Instead, use the `Color.FromArgb` method.

The fourth example uses the `Color` property of the `SolidColorBrush` class. The last example in the listing is a bit offbeat. In that example, you use the `Windows.UI.Xaml.Markup.XamlReader` class to load in a string of XAML. In this way, you can parse a color hex string or any other chunk of XAML that resolves down to a brush. This can be a great approach to use when storing resources in a database or user-supplied configuration files.

It may seem ridiculous to have so many ways of specifying a color in XAML and in code, but there's good reason for them, as you'll see when we look at gradients.

7.1.2 Gradient brushes

A gradient is a smooth transition between one or more colors. This may be as simple as the old blue-to-black diagonal transition on late '90s PowerPoint presentations, ranging to something that visually represents the night sky, complete with a hard break at the horizon line. In general, the subtle use of a gradient can provide a sense of depth to an interface. In contrast, figure 7.2 is a completely nonsubtle gradient going from white to black and back to white.

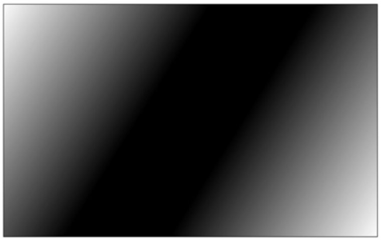


Figure 7.2 A white-to-black-to-white gradient on the diagonal. The gradient is used to fill a black-bordered rectangle.

The Windows 8 design aesthetic may generally favor solid colors, but that doesn't mean a well-placed gradient can't be both useful and visually appealing. Before going off and filling your UI with gradients, however, consider that they do add a very slight tax on the rendering system (more so when animated). Also consider that the gradient should be very subtle and serve some useful purpose in order to stay within the design guidelines.

The following listing creates the gradient shown in figure 7.2.

Listing 7.3 The white/black/white gradient from the start of this section

```
<Grid x:Name="LayoutRoot" Background="White">
  <Rectangle Width="800" Height="500" Stroke="Black">
    <Rectangle.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
        <LinearGradientBrush.GradientStops>
          <GradientStop Offset="0" Color="White" />
          <GradientStop Offset="0.4" Color="Black" />
          <GradientStop Offset="0.6" Color="Black" />
          <GradientStop Offset="1" Color="White" />
        </LinearGradientBrush.GradientStops>
      </LinearGradientBrush>
    </Rectangle.Fill>
  </Rectangle>
</Grid>
```

Brush with start and end points

Gradient stops

The `StartPoint` and `EndPoint` properties, illustrated in figure 7.3, control the orientation of the gradient. In this way, you can position the gradient on the horizontal, vertical, or diagonal or through fractional/double values, any place in between.

Figure 7.3 shows the gradient coordinate system, as well as an illustration of this specific gradient and an approximation of its parameters. You can see that the grid is based on floating-point values between 0 and 1, with 0.5 being the halfway point. Using relative (U/V style for the 3D enthusiasts) coordinate mapping makes the gradient agnostic of physical resolution.

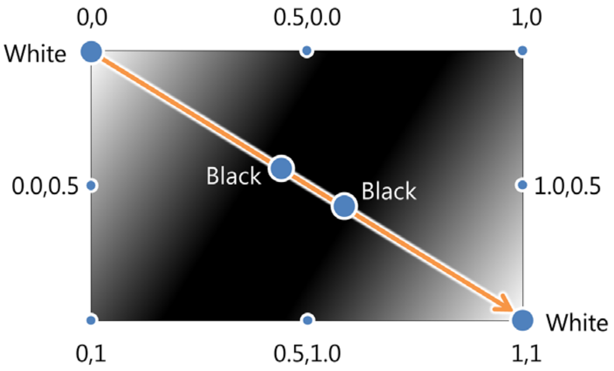


Figure 7.3 The linear gradient with the stops explained. The stops are 0.0: white, 0.4: black, 0.6: black, and 1.0: white. The gradient start point is 0,0 and end point is 1,1.

Like everything else in XAML, you can also create the `LinearGradientBrush` completely from within code. The next listing shows how to do this, assuming the page's XAML includes a rectangle with the name "rect."

Listing 7.4 `LinearGradientBrush` created in code

```
public MainPage()
{
  this.InitializeComponent();
  FillRectangle(rect);
}

private void FillRectangle(Rectangle r)
{
  var brush = new LinearGradientBrush();

  brush.StartPoint = new Point(0,0);
  brush.EndPoint = new Point(1,1);

  for (int i = 0; i < 4; i++)
    brush.GradientStops.Add(new GradientStop());

  brush.GradientStops[0].Offset = 0;
  brush.GradientStops[0].Color = Colors.White;

  brush.GradientStops[1].Offset = 0.4;
  brush.GradientStops[1].Color = Colors.Black;

  brush.GradientStops[2].Offset = 0.6;
  brush.GradientStops[2].Color = Colors.Black;

  brush.GradientStops[3].Offset = 1.0;
  brush.GradientStops[3].Color = Colors.White;

  r.Fill = brush;
}
```

Set start and end points

Create stops

Assign colors and offsets

Paint the rectangle

If you run this listing, you'll see exactly the same result as from listing 7.3. I encourage you to do as much of your UI as possible using markup. But when you want to dynamically create UI elements or need to share a UI design with other XAML platforms, code is often the easiest approach.

Although I didn't show it here, you can dynamically load XAML from the code-behind just as in the `SolidColorBrush` example.

> **WHERE IS THE RADIAL GRADIENT BRUSH?** You won't find the `RadialGradientBrush` in WinRT XAML. The reason is that a radial gradient generally goes against the design aesthetic. (These brushes were almost always used for spherical highlights.) The team wanted to discourage its use and make the most use of their available development time by concentrating on the essentials. If you really need a radial gradient, you'll need to load in a PNG with one prerendered or render it as part of a DirectX surface in C++.

You can do quite a bit with a gradient and enough stops. The more stops you have, the more processing time will be consumed by calculating the steps, so there's a point in design when you're better off using another approach. Sometimes you just need to paint with something a little more complex than a gradient. For those occasions, consider using the `ImageBrush`.

7.1.3 Image brushes

The final brush we'll discuss in this chapter is the `ImageBrush`. Unlike the `SolidColorBrush` and `LinearGradientBrush`, the `ImageBrush` paints not with individual colors but with an entire image.

Use an `ImageBrush` when you need to fill an arbitrary shape with an image. Also consider using an `ImageBrush` instead of an `Image` element (covered later in this chapter) in cases where you need to show the same image a number of times on the same page, because the `ImageBrush` will generally perform better in that scenario.

I used an interesting photo in figure 7.4, but you're not likely to find that type of `ImageBrush` use in top apps in the Windows Store. Yes, it was popular back in the day to paint fonts with images, but that generally doesn't fit well in the Windows design aesthetic. Instead, you can use the `ImageBrush` to provide shaped clipping to images. For example, you may want photos to show up in a circular or oval frame. You can paint the image to an oval in that case.

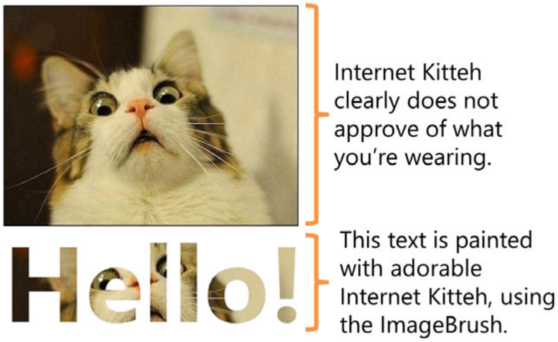

Internet Kitteh clearly does not approve of what you're wearing.
Hello!
This text is painted with adorable Internet Kitteh, using the ImageBrush.

Figure 7.4 `ImageBrush` being used to paint the itteh bitteh Internet Kitteh's shocked face all over our UI

The following listing shows how to create the image shown in figure 7.4. It assumes there is an image in the project's Assets folder, marked as content and named shocked.jpg.

Listing 7.5 Painting shapes and text using an `ImageBrush` in XAML

```
<StackPanel Margin="50">
  <Rectangle x:Name="rect" Width="400" Height="300" Stroke="Black">
    <Rectangle.Fill>
      <ImageBrush ImageSource="/Assets/shocked.jpg"
                  Stretch="UniformToFill" />
    </Rectangle.Fill>
  </Rectangle>

  <TextBlock x:Name="HelloText" Text="Hello!" Width="400"
             FontSize="145" FontWeight="Bold">
    <TextBlock.Foreground>
      <ImageBrush ImageSource="/Assets/shocked.jpg"
                  Stretch="UniformToFill"/>
    </TextBlock.Foreground>
  </TextBlock>
</StackPanel>
```

Paint Rectangle with image

Paint TextBlock with image

This example shows how to fill both a rectangle and text using the same source image. The `Stretch` property will be covered later in this chapter when we cover the `Image` element. For now, understand that it controls how the image is stretched and scaled to fit the space to be painted.

The next listing shows the equivalent to listing 7.5 but in code.

Listing 7.6 Painting shapes and text using an `ImageBrush` in C#

```
public MainPage()
{
  this.InitializeComponent();
  PaintElements();
}

private void PaintElements()
{
  ImageBrush br = new ImageBrush();
  br.ImageSource =
    new BitmapImage(new Uri("ms-appx:/Assets/shocked.jpg"));

  rect.Fill = br;
  HelloText.Foreground = br;
}
```

BitmapImage source using ms-appx protocol

Paint Rectangle

Paint TextBlock

I wanted to show this example because it shows the URI format for appx-embedded resources. Note the `ms-appx:` prefix for the resource. This means to look inside the installation itself, not out on the web. This approach differs from the often confusing approach used in Silverlight where a leading slash decides between local or remote resources. It's also easier to remember than the slash-heavy WPF pack URI syntax.

Note also that you have to use a `BitmapImage` class in this case. When in XAML, the translation into a `BitmapImage` is done for you automatically by a type converter. When in code, it's up to you to set this.

You'll use the `SolidColorBrush` more often than any of the other available brushes. The `LinearGradientBrush` is also very nice, but be careful how you apply it—resist the temptation to create gloss and shadows on your UI. With either brush, there are a number of different ways to specify the colors, both in code and in markup. For the most part, you'll find yourself using the hex color codes, just as you would in HTML and CSS.

The `ImageBrush` is a great way to step outside the bounds of what solid colors and gradients can provide. If there's no clean way with the other brushes to achieve the result you're looking for, simply create what you want in Photoshop and load it in with an `ImageBrush`.

You may have noticed that the default `Grid` background color in the projects is specified using a `{StaticResource}` markup extension. In those cases, the brush was stored as a resource.

7.2 Resources

Cascading Style Sheets (CSS) make it possible to reuse styles, colors, and more in HTML. In XAML, we've been defining all of our object properties at the object level. If you follow that approach, and you decide to change the color scheme for your app, you'll have a lot of manual searching and replacing to do. The same is true in HTML: If you define all your colors and styles locally, it makes reskinning the site much harder.

In XAML, reuse of colors and styles is handled through *resources*. Resources are reusable instances of types. Typically, resources are things like brushes or styles and templates. They can also be class instances, viewmodels (covered in chapter 9), and more. Almost any class can be a resource, but because the objects in a resource dictionary must be sharable, most visual elements cannot. Sharing not only helps with reuse, but it can be a memory saver as well, because you have more control over the number of objects in use.

You can declare resources from code, but the more common approach is to declare them in markup. In this way, you can define reusable objects that exist within an element's scope, on a page, app-wide, or through the use of dictionaries, or any combination of those scopes.

All resources are added to a special type of lookup table called a *resource dictionary*. In some cases, this is transparent to you. In others, you explicitly create separate resource dictionary files. In all cases, when using resources from markup, you do so using the `StaticResource` markup extension.

In this introduction to resources, I'll focus on simple resources: brushes. Starting simple here makes it easy to understand this important concept when it is used in many of the subsequent chapters—especially when we discuss styles and templates. First, we'll look at resources defined locally and on a single page. Then, we'll look at the common practice of defining app-wide resources. Finally, we'll look at the concept of a resource dictionary, something the built-in templates make extensive use of for standard control styles and colors.

7.2.1 *Local and page resources*

Resources may be used within the scope in which they're defined. If you define a resource at the `Grid` level, only elements inside the `Grid` will see it. If you define the resource at a `ListBox` level, only the children of the `ListBox` will see it.

Because of this, one common place to define a resource is at the `Page` or `UserControl` level. That way, the resource works for every element inside that `Page` or `UserControl`, including the `Page` or `UserControl` itself.

The following listing shows how to define and use a page-level resource and a grid-level (local) resource in XAML.

Listing 7.7 Page and local resources defined and used in markup

```
<Page
  x:Class="ResourceExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ResourceExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Page.Resources>
    <LinearGradientBrush x:Key="StandardGradient"
                         StartPoint="0,0" EndPoint="1,1">
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0" Color="White" />
        <GradientStop Offset="0.4" Color="Black" />
        <GradientStop Offset="0.6" Color="Black" />
        <GradientStop Offset="1" Color="White" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel x:Name="StackPanel1">
      <StackPanel.Resources>
        <SolidColorBrush x:Key="ItemBackground"
                         Color="White" />
      </StackPanel.Resources>

      <Grid Height="300" Margin="20"
            Background="{StaticResource ItemBackground}">
        <Rectangle Fill="{StaticResource StandardGradient}"
                   Margin="50"/>
      </Grid>

      <Grid Height="300" Margin="20"
            Background="{StaticResource ItemBackground}">
        <Rectangle Fill="Purple" Margin="50"/>
      </Grid>
    </StackPanel>
  </Grid>
</Page>
```

Page-level resources

Resource StandardGradient

Local resources

Resource ItemBackground

Use page-level resource

Use local resource

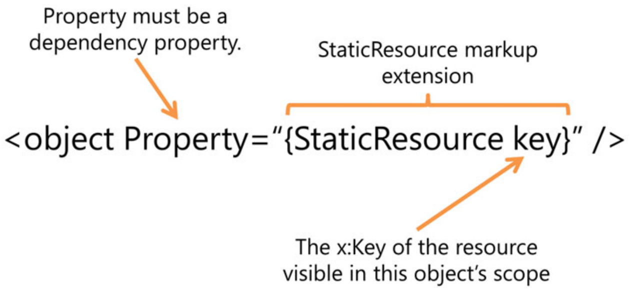

Property must be a dependency property.
StaticResource markup extension
<object Property="{StaticResource key}" />
The x:Key of the resource visible in this object's scope

Figure 7.5 A breakdown of the components of the `StaticResource` markup extension as used to assign a property value. The extension is a helpful way of specifying the longer-form `<object><object.Property><StaticResource ResourceKey="key">...` value.

All resources must have a key. That key must be unique within any combined scope so that no two resources visible to an item have the same key. That is, don't define a page-level resource and, say, an app-level resource with the same key.

Of course, the terms *local* and *page-level* are simply conveniences. In reality, resources simply have scope, and whether you consider something local really depends on which element's relationship you're looking at.

In this example, resources are defined both at the page level and at a level local to the `StackPanel` named `StackPanel1`. Only elements on the page may use the page-level resource `StandardGradient`. Similarly, only the `StackPanel` and child elements of it may use the `ItemBackground` resource.

When using a resource, always use the `StaticResource` markup extension. The format for using this extension is illustrated in figure 7.5.

Note that in figure 7.5 I specifically called out that the property must be a dependency property. This is important: Only dependency properties (covered in chapter 3) can derive their value from a resource or from an animation. If you run into an error using a resource with a property, double-check to make sure the property is a `DependencyProperty`.

When you want to access resources from code, the `StaticResource` markup extension doesn't come into play. Instead, you simply refer to the resource by its key and get back an appropriate type, as shown here.

Listing 7.8 Accessing page and local resources from code

```
public MainPage()
{
  this.InitializeComponent();

  var resource = this.Resources["StandardGradient"] as Brush;

  var localRes = StackPanel1.Resources["ItemBackground"] as Brush;
}
```

Page resource

StackPanel-level local resource

You can add resources from code, although that's very rarely done. Just make sure you add the resources before the visual tree with elements containing references to the resources is loaded. Typically, this is going to be in the constructor, before the `InitializeLayout` function call. The `Loaded` event (which fires after completion of `InitializeLayout` and page loading) is too late to add resources.

Putting resources on a page is a common approach to reusing styles, brushes, and more at only the page level. Resources at levels below that (local resources) are usually confined to data templates (covered in chapter 9) just because of a desire to generally keep resources together and easily identified.

Scoping of page and local resources is pretty easy to see. There's one additional scope beyond that, however, that may not be immediately obvious.

7.2.2 Application resources

By far, the most common resource management approach used by app developers is to put the resources in app.xaml. By defining the resources there, they're available to the entire app and may be used in any `Page` or `UserControl`.

Just by virtue of the fact that the resources are defined in app.xaml makes them app-global in scope. Should you decide to promote a page-level resource to an app-level resource, all you need to do is cut and paste the resource itself—everything else stays the same. The next listing shows some resources defined at the application level.

Listing 7.9 Application resources

```
<Application
  x:Class="ResourceExample.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ResourceExample">

  <Application.Resources>
    <ResourceDictionary>
      <SolidColorBrush x:Key="AppTextColor" Color="White" />
      <SolidColorBrush x:Key="AppTextBackground" Color="Black" />

      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Regular resources

Standard resource dictionary

The `ResourceDictionary` tag is optional when you have only standard keyed resources. But as you'll see shortly, if you plan to merge in any other resource dictionaries, it's required.

Resource dictionaries outside of app.xaml don't start with the `Application` tag. Instead, they start directly with the `ResourceDictionary` tag. Once inside that tag, everything else is the same as it is in app.xaml.

7.2.3 Resource dictionaries

Just as you wouldn't want a single code file with thousands of lines of code, you don't want enormous XAML files. Once the number of resources gets large (where *large* is pretty subjective) you'll want to put them into one or more separate resource dictionaries. Doing so enables you to break them up by groupings that are logical to your application as well as to your team. You'll get more manageable source control and so on.

More important, resource dictionaries enable you to be more thoughtful about which pages merge in which resources. If, for example, you have a set of resources that's used in only 10% of the pages in the app, making those resources available app-wide is a waste of memory. Not only that, but loading resources exacts a performance penalty—the more resources you have defined in (or merged into) app.xaml, the longer your app will take to load.

You can create a new resource dictionary by simply adding a new file using the Resource Dictionary template. But every project already has several resource dictionaries of interest. One resource file you may not immediately notice (because it's in the Windows SDK location, not the local project) is the ThemeResources.xaml file, a small portion of which is shown here.

Listing 7.10 An example resource dictionary

```
<ResourceDictionary
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ResourceDictionary.ThemeDictionaries>
    <ResourceDictionary x:Key="Default">
      <FontFamily
         x:Key="ContentControlThemeFontFamily">Segoe UI</FontFamily>
      <FontFamily
         x:Key="SymbolThemeFontFamily">Segoe UI Symbol</FontFamily>
      <x:Double x:Key="AppBarThemeMinHeight">68</x:Double>
      <Thickness x:Key="TextControlThemePadding">10,3,10,5</Thickness>
      <Thickness x:Key="ToggleButtonBorderThemeThickness">2</Thickness>
      <Thickness x:Key="ToolTipBorderThemeThickness">2</Thickness>
      <SolidColorBrush x:Key="AppBarBackgroundThemeBrush"
                       Color="#E5000000" />
      <SolidColorBrush x:Key="AppBarBorderThemeBrush"
                       Color="#E5000000" />
      <SolidColorBrush x:Key="AppBarItemBackgroundThemeBrush"
                       Color="Transparent" />
      <SolidColorBrush x:Key="AppBarItemDisabledForegroundThemeBrush"
                       Color="#66FFFFFF" />
      <SolidColorBrush x:Key="AppBarItemForegroundThemeBrush"
                       Color="#FFFFFFFF" />
      <SolidColorBrush x:Key="AppBarItemPointerOverBackgroundThemeBrush"
                       Color="#21FFFFFF" />
      <SolidColorBrush x:Key="AppBarItemPointerOverForegroundThemeBrush"
                       Color="#FFFFFFFF" />
      ...
    </ResourceDictionary>
    <ResourceDictionary x:Key="HighContrast">
      ...
    </ResourceDictionary>
  </ResourceDictionary.ThemeDictionaries>
</ResourceDictionary>
```

Theme support

Default theme

Key system parameter

High-contrast theme

There are three important things to notice in this listing:

- Resource dictionaries can themselves have keys.
- Resources aren't limited to brushes.
- Most critical system UI parameters exist as resources.

Resource dictionaries can have keys specifically to support theming. We'll discuss themes in more detail in the next chapter when I also cover styling. Note that currently the only system themes supported are standard and high contrast.

The second thing to notice is that the resource files contain much more than just brushes. You'll see not only control styles but also simple types like `Font`, `double`, and `Thickness`. Resources can be used to standardize these across the app.

Finally, when designing your own custom controls and page layouts, it will help if you know the contents of the theme resources file. Make use of the built-in sizing and other constant resources whenever possible.

One unique aspect of stand-alone resource dictionaries is that they can be merged into other dictionaries, providing access to the resources defined therein.

> **THE STANDARD RESOURCES** In addition to the StandardStyles.xaml in the Common folder of every new project, there are two other important XAML resource dictionaries. First is the ThemeResources.xaml shown in listing 7.10. This handles dark and light theme color and style settings. The second is generic.xaml, which includes the Windows 8-style UI templates for the built-in controls. Both resource dictionaries can be found in your Program Files (x86) folder, under \Windows Kits\8.0\Include\winrt\xaml\design. These resources are for your education and to help designers; changing them will not necessarily alter the built-in styles at runtime. These files are updated with the Windows SDK.

MERGING RESOURCE DICTIONARIES

Resource dictionaries (whether stand-alone or the `Resources` property of any `FrameworkElement`) can include, or "pull in," other resource dictionaries. This is called merging.

Each resource dictionary must merge in any other resource dictionaries it relies on. It's not sufficient for the resource to simply be defined ahead of time; it must be merged in. Think of it like include files in C (if you're familiar with that). Look at each resource dictionary file individually and ensure that it has merged into it all the other resource dictionaries it requires. The XAML resource management system will ensure the same resources aren't physically stored or created multiple times.

The following listing shows the StandardStyles.xaml resource file merged into app.xaml. Every XAML project includes at least this set of merged-in styles to provide common colors, app bar buttons, and more.

Listing 7.11 The standard resource dictionary merged at the application level

```
<Application
  x:Class="ResourceExample.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ResourceExample">

  <Application.Resources>
    <ResourceDictionary>
      <SolidColorBrush x:Key="AppTextColor" Color="White" />
      <SolidColorBrush x:Key="AppTextBackground" Color="Black" />

      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Regular resources

Standard resource dictionary

In this example, there are two standard resources and one merged-in resource dictionary. The StandardStyles.xaml can't see the `AppTextColor` or `AppTextBackground` resource; if it needs them, they must be defined in that file, or they must be moved to a separate resource dictionary and merged into the StandardStyles.xaml. All of these resources here can be seen app-wide because they're defined in or merged into the resource dictionary in app.xaml.

Earlier, I mentioned that every resource incurs a load time penalty. So, when you create your final versions of your app, you should remove from StandardStyles.xaml any resources your app isn't using. To verify that you're removing the correct ones, comment them out and run through your tests. Once you're certain, remove the resources to cut down on file size. Should you need additional standard resources in the future, you can copy them in from another project.

Let's say that you're creating something highly custom, like a game. In that game, only a couple of opening screens use the standard styles—everything else is custom drawn on multiple other pages. In that case, you may want to merge the standard styles into only the first couple pages. The next listing shows how to merge in a resource dictionary at the page level.

Listing 7.12 Merging a resource dictionary at the page level

```
<Page
  x:Class="ResourceExample.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ResourceExample"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Page.Resources>
    <ResourceDictionary>
      <LinearGradientBrush x:Key="StandardGradient"
                           StartPoint="0,0" EndPoint="1,1">
        <LinearGradientBrush.GradientStops>
          <GradientStop Offset="0" Color="White" />
```

Page resources

Gradient brush resource

```
          <GradientStop Offset="0.4" Color="Black" />
          <GradientStop Offset="0.6" Color="Black" />
          <GradientStop Offset="1" Color="White" />
        </LinearGradientBrush.GradientStops>
      </LinearGradientBrush>

      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Page.Resources>
...
</Page>
```

Merged-in resource dictionary

As before, I'm showing both standard resources as well as resource dictionaries together. The reason is, without seeing the syntax (that both must be in the `ResourceDictionary` tag), it can appear that you're unable to mix and match. I want to assure you that you can. You'll notice that the approach here is the same as used in app.xaml.

Regardless of where you define them, resources are a great way to centralize the definitions of colors, fonts, brushes, sizes, and much more in XAML. In this chapter, I focus specifically on brushes, but throughout the rest of the book, you'll see resources used primarily for styles and control templates. Let's look at styles and how they are expressed as resources.

7.3 Styles

When creating the UI for an app or a bit of content, you're encouraged to be consistent in the use of font faces, font sizes, colors, spacing, and more. Most modern UI and document markup languages provide some way to establish and reuse visual properties of elements. In HTML, it's a style defined using CSS. In Microsoft Word, it's a style, and in XAML it's also called a style.

When compared to other markup languages, XAML takes a slightly different approach to styles. In XAML a style is simply a set of property values—any property, as long as it's a dependency property. This means that you can define styles that represent not only the typical properties like colors and margins but even values like `Text`, `Content`, and the control templates.

In this section, I'll first introduce you to the concept of an explicit or keyed style. This is a style that you must reference using the `StaticResource` markup extension. Then, I'll show how styles can be inherited, much like they can in CSS. Finally, I'll demonstrate implicit styles: styles that don't require a `StaticResource` reference. Styles are most easily understood when used naturally. Rather that show an example of every possible use of styles, I'll cover the important points here and then use styles, and explain that use, throughout the rest of this chapter and the rest of the book.

7.3.1 Explicit or keyed styles

When designing the UI, you usually end up with elements that are based on the same type but that must be styled differently. For example, you may have a `TextBlock` that's used for heading text and another `TextBlock` that's used for body text. In CSS, you'd typically accomplish this by using a named CSS class and then referencing it in markup. XAML is similar—if you think of the resource key as that class name, you'll get the main idea.

As you learned in the previous section, resource dictionaries require a key for each entry in the dictionary. An explicit style is a style that, in addition to the type specification, has a key that may then be referenced by the type using the `StaticResource` markup extension. The following listing shows an explicit style in action.

Listing 7.13 Using an explicit style to set properties for several elements

```
<Grid Background="White">
  <Grid.Resources>
    <Style TargetType="TextBlock" x:Key="HeaderStyle">
      <Setter Property="FontFamily" Value="Segoe UI" />
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="FontSize" Value="42" />
      <Setter Property="Margin" Value="10 30 10 20" />
    </Style>
    <Style TargetType="TextBlock" x:Key="BodyStyle">
      <Setter Property="FontFamily" Value="Segoe UI" />
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="Margin" Value="10 0 10 10" />
    </Style>
  </Grid.Resources>

  <StackPanel>
    <TextBlock Text="Heading"
               Style="{StaticResource HeaderStyle}"/>
    <TextBlock Text="I'm some body text"
               Style="{StaticResource BodyStyle}"/>
    <TextBlock Text="So am I"
               Style="{StaticResource BodyStyle}" />
  </StackPanel>
</Grid>
```

Define HeaderStyle

Define BodyStyle

Use HeaderStyle

Use BodyStyle

This listing defines two explicit styles and uses them in the `TextBlock` elements in the main `StackPanel`. The styles are defined as resources at the `Grid` level. Because the `TextBlock` elements can trace a path back up to the `Grid` in the tree of elements, they can use the resources. The styles could also have been defined at the `StackPanel` level.

The styles in use here not only make it easy to centralize the definition of properties, but they also avoid needless repetition of properties, making the main control tree much easier to understand when looking at the XAML.

Once you have an explicit style, you can also use it as the basis for style inheritance, a great way to consolidate style properties in a large project.

7.3.2 Style inheritance

Consider the typical app: It has the same font family and foreground color used throughout all the text, but it has several different sizes of text, margins values, and more in use throughout the pages. Some of those may be headings, others may be body text, and others may be field labels. Ideally, you don't want to have to copy the core properties to every style; you want to reference those core properties so you can change them in the least number of places as possible. The next listing shows an interesting example of using style inheritance.

Listing 7.14 Style inheritance

```
<Grid Background="White">
  <Grid.Resources>
    <Style TargetType="TextBlock" x:Key="BaseTextStyle">
      <Setter Property="FontFamily" Value="Segoe UI" />
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="Margin" Value="10 0 10 10" />
    </Style>
    <Style TargetType="TextBlock" x:Key="HeaderStyle"
           BasedOn="{StaticResource BaseTextStyle}">
      <Setter Property="FontSize" Value="42" />
      <Setter Property="Margin" Value="10 30 10 20" />
    </Style>
    <Style TargetType="TextBlock" x:Key="BodyStyle"
           BasedOn="{StaticResource BaseTextStyle}" />
  </Grid.Resources>

  <StackPanel>
    <TextBlock Text="Heading"
               Style="{StaticResource HeaderStyle}"/>
    <TextBlock Text="I'm some body text"
               Style="{StaticResource BodyStyle}"/>
    <TextBlock Text="So am I"
               Style="{StaticResource BodyStyle}" />
  </StackPanel>
</Grid>
```

Define BaseTextStyle

Define HeaderStyle based on BaseTextStyle

Define BodyStyle based on BaseTextStyle

Use HeaderStyle

Use BodyStyle

In this example, you accomplish the same styling as shown in listing 7.13 but use inheritance. The result is that the core properties, like the font family and foreground colors, are now defined in a single location.

This example shows a pretty straight forward example of style inheritance—or does it? Take a look at the `HeaderStyle` static resource. Notice how it overrides two of the properties of the base style. That's something that's easy to do using style inheritance. You can even take it to extremes, as in the `BodyStyle` resource. That resource defines no setters of its own; it simply inherits everything from the base style. But because it has been explicitly named, you'll be able to add property setters to it in the future and know the UI will light up with those changes.

I generally recommend that you discourage the direct use of your base styles. Instead, tell the designers and developers on your team to use the other named styles instead, creating new ones if necessary. This way, you won't have to walk through code changing style resource references when you decide to standardize on a new style for a semantic use.

Unfortunately, there's no effective way to enforce this except for manually checking the markup (or using a tool to do this) and punishing those who stray from your guidance. This may even be a great place for you to learn to use something like the .NET Micro Framework to create a device that uses electricity or pointy things to…encourage compliance.

So far, the styles have had to be pulled in by referring to the resource by name. That's not always the most convenient way to accomplish styling an entire app. That's why we also have implicit styles.

Control templates

Once you get XAML styling under your belt, you'll want to learn how to modify the control templates. If you look at the default style for a control (this can be easily done in Expression Blend), you'll see that it includes a control template. This template contains all the parts that make up the control's appearance.

A control's code is responsible just for behavior. It is the control template that completely defines the visual representation of a control.

Think of the style as a set of property values and the template as the set of UI elements that make up a control. You create the control template but then, through the use of `{TemplateBinding}` references, refer back to the properties provided by the control and set in the default style. In this way, you can set the `Background` property of the rectangle that makes up the background to the `Background` property of the control.

The control template is just XAML, but it's long enough that reproducing it here isn't going to do anyone any favors. You can modify control templates by pasting in the XAML from the generic.xaml for the control and manually tweaking what's in there, but I don't recommend that. Instead, it's time to finally crack open Expression Blend and choose the menu option to modify the default template for the control.

Depending on how well the template is implemented, you may be able to do all the customization you need just by using a style. For the other times, create a new control template, put it in a named or (as you'll see shortly) implicit style, and store it in a resource dictionary or app.xaml.

7.3.3 *Implicit styles*

I don't know about you, but I get tired of typing `{StaticResource SomeStyleKey}` over and over again in the pages in my app. It's not exactly a hardship, but I'd rather just define the style in a central location and have it picked up by every instance of that type throughout the page and the app as a whole.

In CSS, this is handled by assigning a style to a specific type of markup element, like a `div` element, an `article` element, or a `p` element. Semantic HTML makes extensive use of this to avoid cluttering the body of the page with style information. This same approach can work in XAML, although we don't have as rich a differentiation between text elements.

Remember how I earlier said that every entry in a resource dictionary must have a key? Well, sometimes that key is implicit, something you don't enter manually. The resource key in an implicit style is the target type itself. Consider the following listing.

Listing 7.15 Using an implicit style

```
<Grid Background="White">
  <Grid.Resources>
    <Style TargetType="TextBlock" x:Key="BaseTextStyle">
      <Setter Property="FontFamily" Value="Segoe UI" />
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="Margin" Value="10 0 10 10" />
    </Style>
    <Style TargetType="TextBlock" x:Key="HeaderStyle"
           BasedOn="{StaticResource BaseTextStyle}">
      <Setter Property="FontSize" Value="42" />
      <Setter Property="Margin" Value="10 30 10 20" />
    </Style>
    <Style TargetType="TextBlock"
           BasedOn="{StaticResource BaseTextStyle}" />
  </Grid.Resources>

  <StackPanel>
    <TextBlock Text="Heading"
               Style="{StaticResource HeaderStyle}"/>
    <TextBlock Text="I'm some body text" />
    <TextBlock Text="So am I" />
  </StackPanel>
</Grid>
```

Define explicit BaseTextStyle

Define explicit HeaderStyle

Define implicit body style

Use header style explicitly

Use body style implicitly

In this example, the style formerly known as `BodyStyle` has been made into an implicit style. Its definition no longer includes an explicit key. When you use this style, the `TextBlock` doesn't need any `Style` property value at all. This is really convenient when you want to style all or most of the elements of a particular type using the same set of property values.

As you can also see in this example, implicit resources can participate in style inheritance but only as the leaf in the style tree. That is, they can inherit from other styles, but nothing can inherit from them. In this example, the implicit style is empty, as it was in the previous example. That's not a requirement, just a point of illustration.

Now that you've learned how to paint using brushes, sometimes exposed via resources, sometimes included in styles, let's look at some shapes to paint. To quote the now embarrassingly aged[1] singer David Lee Roth, "I've got my pencil! Now give me something to write on."

---

[1] I dare you to watch Mr. Roth in some modern Van Halen video. Some things are better remembered.

7.4 Vector graphics

Although it has great support for bitmapped images, XAML is generally a vector-based system. That is, UI elements are usually defined using shapes with points and brushes. Almost every stock UI element is defined this way: rectangles for buttons, for example. The reason for this is that vector-based elements can be infinitely scaled to accommodate any display resolution without (except in the case of shrinking them to a thumbnail) reduction in quality. You can stretch them anamorphically and change their aspect ratio, you can rotate them, and even distort the shape, all without a loss of quality.

In XAML, all vector graphics types derive from the common `Shape` base class. This class provides properties for handling the brushes and stroke, and through the underlying base classes, alignment and more.

In this section, we'll start at the beginning: lines. From there, we'll create simple shapes, then much more complex polygons. Finally, we'll take a look at the `Path` type and its associated mini language. Throughout, we'll use the solid-color brushes you learned about previously in this chapter.

7.4.1 Line

A line is a point-to-point vector with no fill. Okay, technically, I'm wrong. Because `Line` derives from `Shape`, it has both stroke and fill brushes. The fill, however, is simply not used. I'm not a fan of useless properties, and if designing it myself, I would have maybe provided a `FilledShape` base class in addition to a stroke-only `Shape` base, but I'm sure there's a reason they went this way in the design. Object-oriented purism aside, the line is easy to use and is surprisingly flexible. Figure 7.6 shows several types of lines.

In this figure, I show four lines, each with the same thickness. The first is a plain line with regular flat end caps. The second has one pointy end and one rounded end. The third is back to plain end caps but is evenly dashed. The last line looks like a bit of sequenced DNA, or perhaps a sloppy computer punch card, with its pattern of dashes and spaces.

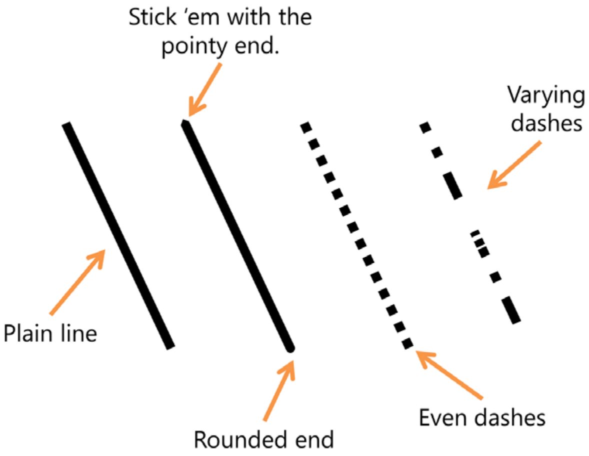


Figure 7.6 Four versions of a plain line, demonstrating different line ends and dash styles

To create these four lines, look to the following listing.

Listing 7.16 Creating lines in XAML

```
<Grid Background="White">
  <Grid.Resources>
    <Style TargetType="Line">
      <Setter Property="StrokeThickness" Value="20" />
      <Setter Property="Stroke" Value="Black" />
      <Setter Property="X1" Value="20" />
      <Setter Property="X2" Value="250" />
      <Setter Property="Y1" Value="10" />
      <Setter Property="Y2" Value="500" />
    </Style>
  </Grid.Resources>

  <StackPanel Orientation="Horizontal" Margin="20">
    <Line />
    <Line StrokeEndLineCap="Round"
          StrokeStartLineCap="Triangle" />
    <Line StrokeDashArray="1" />
    <Line StrokeDashArray="1,2,1,2,3,4,0.5,0.5,0.75,0.25" />
  </StackPanel>
</Grid>
```

Style defining stroke and coordinates

Round end cap

Pointed end cap

Even dashes

Varying dashes

For each line, you provide start `X1`,`Y1`, and end `X2`,`Y2` positions, relative to the line's bounding box in a panel, accounting for margins or any `Canvas.Left` and `Canvas.Top` values. Most of the rest of the code in the example is easy to understand, with the notable exception of the `StrokeDashArray`. In XAML, the array is a set of comma- or space-delimited floating-point numbers. Each number corresponds to a dash or a space and is represented as a multiple of the line's thickness. For example, 1.0 is exactly the line's thickness, 0.5 is half, 2.0 is double, and so on.

The array can be as simple as a single number representing both the dash and space width, all the way through to a very complex list of numbers. In all cases, when the pattern is exhausted, it repeats. Figure 7.7 shows several complex patterns, their visual representation, and the relationship between the spaces and dashes. Note the special case where an odd number of numbers is in the array (3 in this case) and how the pattern alternates.

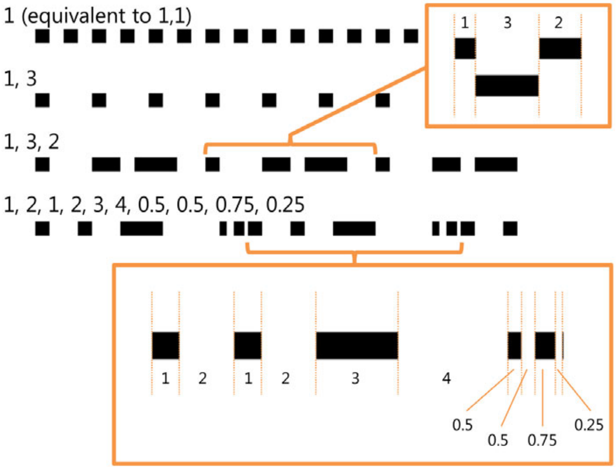


Figure 7.7 The `StrokeDashArray` patterns illustrated. Note both the complex pattern (last line) and the pattern with an odd number of items in the array (third line).

In addition to the `StrokeDashArray` property, you can alter the layout of the line by using the `StrokeDashOffset`. This value, which is expressed in multiples of the line width, controls how far down the line the dash effect starts. Finally, there's the `StrokeDashCap` property, which controls the end caps of each of the dashes.

7.4.2 Polyline

Lines can be useful by themselves, but often they are used to create complex shapes where the start of one line is connected to the end of another. In those cases, the `Polyline` is the shape to use.

Rather than provide discrete X,Y coordinate properties as used in `Line`, the `Polyline` type exposes a `Points` collection. This collection also has a helpful string format, or you can add instances of the `Point` type from code.

The next listing shows how to use the `Points` collection from XAML. It also shows several other properties that we'll discuss after the listing.

Listing 7.17 Creating a polyline in XAML

```
<StackPanel Orientation="Horizontal" Margin="40">
  <StackPanel.Resources>
    <Style TargetType="Polyline">
      <Setter Property="StrokeThickness" Value="5" />
      <Setter Property="Stroke" Value="Black" />
    </Style>
  </StackPanel.Resources>
  <Polyline
     Points="10,10 150,200 200,10 250,300 100,20 75,200 250,135"
     />
  <Polyline
     Points="10,10 150,200 200,10 250,300 100,20 75,200 250,135"
     Fill="Red" FillRule="EvenOdd" />
  <Polyline
     Points="10,10 150,200 200,10 250,300 100,20 75,200 250,135"
     Fill="Blue" FillRule="Nonzero" />
</StackPanel>
```

Implicit style

EvenOdd fill rule

Points

Nonzero fill rule

The `Points` string can use commas or spaces as a delimiter. I prefer to use commas between X and Y components, then spaces to separate the coordinate pairs. This makes it easier to read.

In addition to `Points`, I've also used the `FillRule` property. Figure 7.8 shows the results of the different `FillRule` values used in listing 7.17. This property controls how the `Polyline` is filled with a color.

Look at the first `Polyline` in the figure, and then put your figure on any random point in the whitespace inside or outside the lines. Now, draw

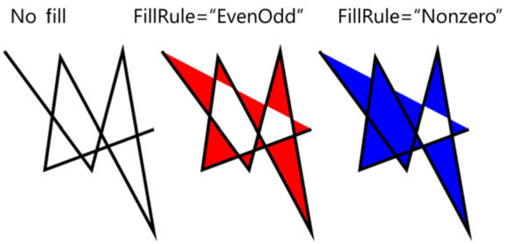


Figure 7.8 Fill rules in a `Polyline`. Note the difference between `EvenOdd` and `NonZero` as well as the implied line between the start and end points.

an imaginary ray (a line with direction) from that point in any direction (just pick a direction) and count how many other lines you cross while doing so. In the case of `EvenOdd`, if the number of lines you cross is odd, the point is considered "inside" the shape and will be filled.

It starts with the same point and arbitrary direction approach, but `NonZero` is otherwise a little more complex. For `NonZero`, you need to know the direction the line segment is going based on its start and end points. Once you know that, you can tell if the line crosses your imaginary ray from left to right or right to left. Each time a line crosses from left to right, add one to your count. Each time it crosses from right to left, subtract one. If the number ends up anything other than zero, the point is considered "inside." MSDN has a ton of great info on this topic if you want to learn about it in more depth.

> **POLYGONS** The `Polygon` shape in XAML is identical to the `Polyline`, except that it automatically closes the shape by connecting the end point to the start point. To do the same with a `Polyline` would require duplicating the first point as the end point. If you're going to do any closing or filling of the shape, I recommend using the `Polygon`.

The `Polyline` type shares all the same stroke- and dash-related properties as the `Line` type and other shapes. Use it when you want to create an open shape made up of many straight-line segments. In addition to the straight-line segments from the `Line` and `PolyLine` types, WinRT XAML supports complex `Path`s made of straight and curved segments.

7.4.3 Paths

A `Path` is a complex line made up of individual straight or curved (or both) segments. `Path`s are primarily used when importing graphics from vector drawing applications like Adobe Illustrator, using AI-to-XAML converters. But you can create `Path`s from scratch in XAML or through native XAML tools like the Microsoft Expression series.

As with everything else in this section, the `Path` type inherits from `Shape`. Because of that, you have access to the same types of brushes and line types as the other shapes. For example, figure 7.9 shows a `Path` made up of several different segments, using brushes and line styles.

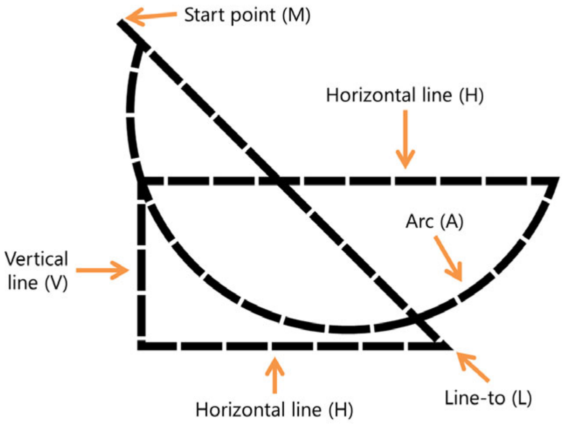


Figure 7.9 **`Path` showing the resulting segments from several different path language commands. Not sure if this is a baby carriage or PacMan preparing for dental work.**

The following listing includes the XAML used to create the `Path` shown in figure 7.9. Note that I could have also specified a fill if I had desired a solid shape.

Listing 7.18 Creating a `Path` in XAML

```
<Grid Background="White">
  <Path Stroke="Black"
     Margin="50"
     StrokeThickness="15" StrokeDashArray="5 0.5"
     Data="M 10,10 L 600,600 H 50 V 300 H 800 A 100,100 75 0 1 50,50" />
</Grid>
```

Path mini language

This listing shows how to create a `Path` using the `Path` mini language in XAML. At first, the language may appear to be complex, but it's actually quite simple. Table 7.1 shows the list of commands available in the mini language.

Table 7.1 The commands in the XAML `Path` mini language. This brings back memories of programming LOGO `Paths` on the Commodore 64 in sixth grade.

Command	Description
`M <startPoint>`	Move. This starts a new figure. Points are specified `x,y`.
`L <endPoint>`	Line between the current point and the specified end point.
`H <x coordinate>`	Horizontal line between the current point and the specified X coordinate.
`V <y coordinate>`	Vertical line between the current point and the specified Y coordinate.
`C <cp1 cp2 endPoint>`	Cubic Bézier curve with control points `cp1` and `cp2`, between the current point and the specified end point.
`Q <cp endPoint>`	Quadratic Bézier curve with control point `cp`, between the current point and the specified end point.
`S <cp endPoint>`	Smooth cubic Bézier curve between the current point and the end point with control point `cp`.
`T <cp endPoint>`	Smooth quadratic Bézier curve between the current point and the end point, with control point `cp`.
A size `rotationAngle isLargeArc sweepDirection` endpoint	Elliptical arc. Draws an elliptical arc with the specified size (x and y radius), rotation angle, current point, and end point. The `isLargeArc` is set to 1 if the angle of the arc is 180 degrees or greater. The `sweepDirection` flag is set to 1 if the arc is drawn in a positive angle direction, 0 if negative.

MSDN has great detailed information on the `Path` language specifics as well. If you plan to create `Path`s from scratch, I encourage you to read through the documents there first.

`Path`s are interesting to learn about and are certainly a useful and powerful way to design vector shapes. But you'll typically run into them only in imported or converted graphics—most apps don't include handwritten `Path` statements. Far more common than the `Path`, or even the `Line` and `Polyline` types, are the `Rectangle` and `Ellipse`.

7.4.4 Rectangles and ellipses

`Line`s are somewhat common—`Polyline`s less so. But the simple `Rectangle` and `Ellipse` shapes show up in UI design again and again. Especially with the Windows design aesthetic, the `Rectangle` is king. Luckily, these shapes are extremely easy to use.

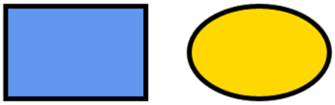


Figure 7.10 A `Rectangle` and an `Ellipse` showing stroke (border) and fill

Listing 7.19 shows how to create the two shapes shown in figure 7.10. In both cases, the shapes are sized by using `Width` and `Height` relative to the top left of the shape, not by discrete points, or in the case of an `Ellipse` or a circle, center point and radii.

Listing 7.19 A `Rectangle` and an `Ellipse` in a `StackPanel`

```
<StackPanel Orientation="Horizontal" Margin="40">
  <Rectangle Width="150" Height="100" Margin="20"
             Stroke="Black"
             StrokeThickness="5"
             Fill="CornflowerBlue" />
  <Ellipse Width="150" Height="100" Margin="20"
           Stroke="Black"
           StrokeThickness="5"
           Fill="Gold" />
</StackPanel>
```

Rectangle

Ellipse

In Windows 8 apps, the `Rectangle` is most often used with a fill but without any stroke. This provides the clean tile look so prevalent in the UI. If you decide to use a stroke, all the same stroke and dash properties available with the line are available here.

Perhaps more common than single-color rectangles in Windows 8 apps is the use of images. We'll wrap up this chapter with coverage of how to get your baby photos, lolcats, memes, or that embarrassing photo of your hair from the '80s (or your parents') on to the app UI.

7.5 Bitmap images

Windows 8 apps are encouraged to be visually rich when showing content. Often, this means that they should make any images front and center. That doesn't mean go and decorate your apps with lots of extraneous imagery, but instead, if images are part of the content, bring them to the forefront. Examples include photos with news stories, video thumbnails, photos from browsing and sharing apps, album covers, and more.

Earlier in this chapter, I introduced the `ImageBrush`. In XAML, the `Image` element is the full element counterpart to that brush. When simply displaying an image, the `Image` element is easier to use, because you don't need to have a separate shape to paint.

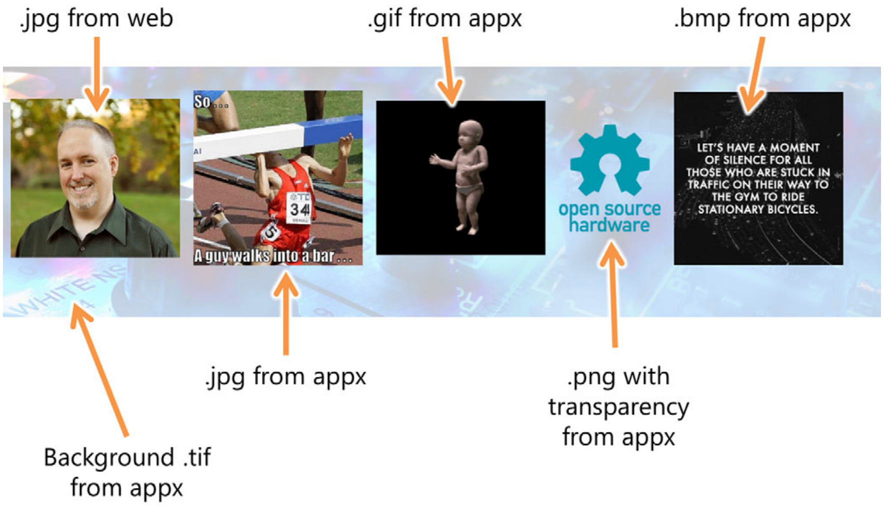

.jpg from web
.gif from appx
.bmp from appx
So...
A guy walks into a bar...
open source hardware
LET'S HAVE A MOMENT OF SILENCE FOR ALL THOSE WHO ARE STUCK IN TRAFFIC ON THEIR WAY TO THE GYM TO RIDE STATIONARY BICYCLES.
.jpg from appx
.png with transparency from appx
Background .tif from appx

Figure 7.11 `Image` **elements displaying images from multiple locations**

Images can come from the local machine, the project, or from the web. Figure 7.11, for example, shows images both from the project as well as a website, in multiple formats.

The formats supported for image processing depend on the codecs available on the machine. By default, these include .bmp, .jpg, .gif, .png, and .tif. Of those, .png is the most commonly used because of its high-quality compression and support for transparency.

The next listing shows how to display images to create the UI shown in figure 7.11. Note the different types of URIs. The `ms-appx` URI format was covered in the `ImageBrush` section of this chapter.

Listing 7.20 Using bitmap images in markup

```
<Grid Background="White">
  <Image Source="ms-appx:/Assets/pmb_youtube_background.tif"
         Opacity="0.25"
         Stretch="UniformToFill"/>
  <StackPanel Orientation="Horizontal" Margin="40">
    <StackPanel.Resources>
      <Style TargetType="Image">
        <Setter Property="Margin" Value="10" />
        <Setter Property="Width" Value="250" />
      </Style>
    </StackPanel.Resources>

    <Image
       Source="http://10rem.net/pub/pmb_fall_2011_color_800px.jpg" />
    <Image Source="ms-appx:/Assets/bar.jpg" />
    <Image Source="ms-appx:/Assets/dancingbaby.gif" />
```

Background TIFF with transparency

JPEG from web

Local JPEG

Local GIF

```
    <Image Source="ms-appx:/Assets/oshw-logo-200-px.png"
           Width="150"/>
    <Image Source="ms-appx:/Assets/bikes.bmp" />
  </StackPanel>
</Grid>
```

Overridden styled property

Local PNG

Local bitmap

This listing shows several different types of images loaded into a `StackPanel`. WinRT takes more after the WPF side than the Silverlight side when it comes to the various formats it supports. We even have GIF support, although I'm sorry to report that the baby is not actually dancing—only the first frame is rendered.

Multi-DPI and contrast images

Windows 8 has better support for high-DPI displays than any version before it. It'll actually be quite common to see high-DPI displays on tablets. XAML will automatically scale your images to the correct resolution but won't give you the best results. Instead, you'll want to provide images at different DPI/resolutions.

Luckily, XAML and WinRT make it easy to use multiple image files. Create the image files at 100%, 140%, and 180% scale. Then name them, respectively, yourimagename.scale-100.png, yourimagename.scale-140.png and yourimagename.scale-180.png.

For high-contrast support, you can optionally add in _contrast-black and _contrast-white, to get, for example, yourimagename.scale-100_contrast-black.png as the filename.

Make sure all the images are in the same folder in the project.

When referencing the image from markup or code, leave out the whole scale bit and simply refer to yourimagename.png. WinRT will automatically pick the correct one.

For more information, see http://bit.ly/Win8MultiDPIImages.

Just as was the case with the `ImageBrush`, you can also create the `Image` element from code, including using the same techniques for specifying the URL. The `Image` element also supports the `ImageFailed` event. This event, which may be wired up from XAML or code, enables you to handle instances where the image was unable to be loaded. In most cases, you'll want to use this to either hide the image or simply display a placeholder graphic from the appx.

Graphics are the building blocks of most visual elements in XAML. They're fundamental, which is why the support is so rich. Not only do you get great support for vector graphics, but you get bitmapped images in a variety of formats as well.

7.6 Summary

This chapter covered brushes, vector graphics, and bitmapped (image) graphics. Brushes can be used to paint the vector graphics and provide borders and shading to most anything. One brush, the `ImageBrush`, crosses the line between vector and bitmapped images by making it possible to paint vector art using an image as the brush.

Because the `Brush` is one of the most commonly used resources, we also covered resources in this chapter. Resources can be local, page-scoped, application-wide, or, through the use of merged-in resource dictionaries, any combination of those scopes. Once you end up with more than a few resources, consider moving them into separate resource dictionaries rather than just keeping everything in app.xaml.

Another very common type of resource is the style. XAML makes it possible for you to define the value for any dependency property using a style setter and then apply that to all elements of a specified type within the scope of that style. You can name styles and use them explicitly, or you can leave the name (key) out and have them implicitly apply to all elements of that type. You can even inherit from a named style to make even better use of common properties.

You won't run into lines and polygons much in most typical WinRT XAML applications, at least not when compared to the more ubiquitous rectangle types. But they are there, and they're exactly the tool you need when you want to draw simple (or complex, in the case of the `Path`) vector art on your UI.

Vectors are great because they scale so well and, when compared to a large bitmapped graphic of the same dimensions, they're typically much smaller in memory footprint. Once you get to very small but complex graphics, a PNG is almost always a more efficient approach. In fact, bitmapped images still have a huge role to play in an otherwise vector-based framework. Use the `Image` element whenever you need to load prerendered content, photos, and more. Even use it to add a little texture to your apps, if you desire.

In the next chapter, we'll take a look at text, a very important part of the Windows 8 UI. You'll use the brush knowledge you gained here to make the text look even more beautiful.

8

Displaying beautiful text

This chapter covers

- Text basics, like the `TextBlock`
- Text wrapping, fonts, alignment, spacing, and more
- Displaying rich and multicolumn text
- Using OpenType font features

Over the years, the use of text in UI design has gone in and out of fashion. Back when I was learning how to program, text and ASCII (and PETSCII) graphics were the mainstay of UIs. Other than the occasional boxed-in border or menu, interfaces were heavily text based. Those poor unfortunate souls toiling away at 12” CRT mainframe terminals at the time rarely had even that—just a sea of amber or green text on a greenish-black glass background, with the only variation being some of it was in reverse.

It was depressing!

Then came Xerox with its GUI (graphical user interface) research, and then the Apple Lisa (which wasn’t successful because it cost more than most cars of the time), the more successful Apple Macintosh, Geos on the Commodore computers and eventually the 286, and, of course, Windows on the PC. These OSs (or shells in some cases) eschewed text and focused more on icons: pictorial representations of elements in the system. Not long after, designers began experimenting with trying to make the UI look more and more like a real-life metaphor. From that you got desktops and the always-good-for-a-laugh Microsoft Bob.

As OSs became more complex and offered more features, the number of icons increased, and the cognitive load associated with remembering all those pictures became a real burden. Most users used a very small percentage of features of their applications because they simply couldn't find everything they needed.

Then Microsoft started experimenting with ways to lighten that load. The Office ribbon was a large part of that. Microsoft had to break a lot of eggs to get the ribbon out there, but after users got used to the change, most agreed it made them more productive. One thing the ribbon did was combine images with a larger emphasis on text. If the display had room, most icons had labels either under them or to the side. Icon groups had labels indicating what you should expect to find in them. The ribbon tabs were identified by their text headings.

You could argue that because of the removal of all-text menus, this was actually a removal of text from the interface. But what really happened was the user saw more text more often, without having to take additional actions.

With Windows 8, we have an even greater reliance on text, not just for commands but for aesthetics and for content. The Windows design aesthetic emphasizes beautiful text above many other elements. Except in the case of games, it's extremely rare to see a Windows app that doesn't have prominent text right on the screen. Sometimes it's a header, sometimes it's simply labels on app bar icons, and often it's incorporated as a key part of the content. In many cases, the typeface used is part of the brand of the app and the corporation or individual who created it.

So, text is important. No, text is really important in Windows 8. Learning how to make the most of text, to make it readable, to make it usable, and to make it beautiful can help you succeed in getting people to use your apps. In this chapter I'm going to show you what text features the platform provides so you can put on your designer hat (or invite your designer over to work with you) and put together an amazing UI for your app.

We'll start the chapter with a look at the basics of text in XAML. First, we'll dive into the most commonly used element, the `TextBlock`. The `TextBlock` is not only popular for displaying text, but it also serves as a great way to see the font and text layout properties in action. As part of our investigation, we'll look at how to combine multiple inline elements to support rich formatting of text. Then, we'll look at how to wrap text and how to align it to the edges or even use full justification. We'll also look at a great feature of the `TextBlock`, the `TextTrimming` property. This property makes it possible to intelligently truncate text and display an ellipsis (…) at the end. This feature is especially useful if you're creating any sort of resizable heading or text layout or localizing text where word or phrase sizes differ significantly from language to language.

We'll wrap up the discussion of the basics by talking about how to control character and line spacing.

The `TextBlock` supports different fonts and colors, but it's more lightweight support than the intended use. To really mix and match fonts, colors, and even embedded UI elements, you need to look to the `RichTextBlock`. This element is also the best one to use to implement multicolumn or more fluid text layout.

One of the most beautiful types of fonts is OpenType. Not only is the normal rendering superior to many other types, but OpenType supports a lot of intelligence about how to format and render alternate capabilities. If you want to automatically convert to small caps, properly format fractions, line up numbers, format for superscript and subscript, or even simulate the classic type styles of historical documents, OpenType makes it possible. We'll dive into some of the great XAML support for OpenType features with an eye toward creating beautiful, compelling, and readable text.

Finally, beautiful text is useless without a guarantee that the right fonts are present. One way to ensure that is to acquire embedding rights and embed the fonts in the application. We'll wrap up the chapter with a look at how to accomplish that.

8.1 Text basics

As you've learned, Windows can handle elements aligned on subpixel boundaries, such as having a Left of 15.76 rather than just 16. This makes layout easier for design professionals and is also essential for smooth animation. It makes the underlying rendering engines, especially for text, more complex, however.

Subpixel layout and rendering apply to text as well. WinRT apps use the ClearType algorithm, provided by DirectWrite (DirectX text stack), to render text using the best quality for a given resolution. ClearType is a specific type of subpixel rendering that uses different colors for the subpixels. On a properly configured display, this results in higher quality rendering than just gray antialiasing. On a poorly configured display, however, ClearType can lead to something that looks like the old NTSC composite text rainbow effects[1] when looking at white text on a black background. Windows supports subpixel rendering and layout of anything, so the text itself may already start on a partial pixel boundary.

Getting the text from the Unicode string and presenting it on displays of varying resolutions using different fonts on different systems is actually fairly complex. It's also a task we only notice when done poorly. The text stack must do the following:

- Read in the source text string.
- Lay out an overall block of text.
- Lay out individual lines within that block.

---

[1] This effect was especially prevalent on the Apple II, or with the Commodore 64 connected to an old TV. If the rainbow effect means nothing to you, ask a parent…or maybe a grandparent…about it.

- Obtain the font information for each character, including combining characters for certain languages.
- Figure out how to display bold and italics (and other styles/weights). There may be a font for it, or it may need to generate pseudo-italic and pseudo-bold text.
- Deal with any text expansion for fonts that support it.
- Lay out individual characters within that line, including subpixel font rendering.
- Render it all out to a rendering surface in hardware.

Any one of those individual steps is a pretty serious programming effort. Luckily, this is primarily handled by the tried and true DirectX text stack. Though all are interesting, the internals of the text stack are pretty well abstracted away from the work you'll normally need to do.

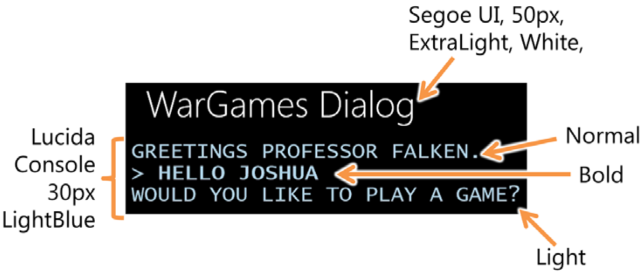
In this section, I'll first introduce you to the `TextBlock`, the most basic text element and the one you'll turn to for most of your text display needs. It's also a convenient element to use to introduce the various font-related properties shared by all text controls.

From there, I'll show you how to enhance the flexibility of the `TextBlock` using inlines. Then, because paragraph formatting is so important, I'll show you how to wrap text and also to truncate text using an ellipsis when the text is too large for the display area. Finally, I'll show you how to control character spacing within a line of text and line spacing within a paragraph of text.

8.1.1 TextBlock

In many apps, the `TextBlock` is the single most common on-page element. This was true even before the Windows 8 emphasis on beautiful and functional text as a core part of the app UI. The `TextBlock` element is so popular because it's incredibly easy to use: Simply set the `Text` property, optionally set the `FontFamily`, `FontWeight`, and `FontSize`, and you have text onscreen as a label, heading, or more. For example, figure 8.1 shows two different fonts in separate `TextBlock` elements, with different font families, weights, and sizes.

Our first listing shows how to create the `TextBlock` elements that display the text shown in figure 8.1. Note the use of different fonts, font sizes, font weights, and foreground colors.



WarGames Dialog

GREETINGS PROFESSOR FALKEN.
> HELLO JOSHUA
WOULD YOU LIKE TO PLAY A GAME?

Segoe UI, 50px, ExtraLight, White,
Lucida Console 30px LightBlue
Normal
Bold
Light

Figure 8.1 **`TextBlock` elements with different font sizes, weights, colors, and font families**

Listing 8.1 Displaying simple text using a `TextBlock`

```
<Grid Background="Black">
  <StackPanel Margin="10">
    <TextBlock Text="WarGames Dialog"
               FontSize="50"
               Margin="20"
               FontWeight="ExtraLight"
               FontFamily="Segoe UI"
               Foreground="White" />

    <TextBlock Text="GREETINGS PROFESSOR FALKEN."
               FontSize="30"
               FontWeight="Normal"
               FontFamily="Lucida Console"
               Foreground="LightBlue" />

    <TextBlock Text="> HELLO JOSHUA"
               FontSize="30"
               FontWeight="Bold"
               FontFamily="Lucida Console"
               Foreground="LightBlue" />

    <TextBlock Text="WOULD YOU LIKE TO PLAY A GAME?"
               FontSize="30"
               FontWeight="Light"
               FontFamily="Lucida Console"
               Foreground="LightBlue" />
  </StackPanel>
</Grid>
```

Segoe UI, 50 px ExtraLight, White

Lucida Console, 30 px, Normal, LightBlue

Lucida Console, 30 px, Bold, LightBlue

Lucida Console, 30 px, Light, LightBlue

This listing shows four `TextBlock` elements with different values for the font-related properties. In WinRT apps, the three you'll use the most are `FontSize`, `FontFamily`, and `Foreground`. `FontWeight` is a runner-up here, because bold display isn't as popular in the Windows design aesthetic.

Not all fonts support all possible font weights. Most will support only a couple of standard weights like `Bold` and `Normal` and will display the closest equivalent for the others. Even the `FontStyle` property (which controls displaying italics) isn't universally supported with different fonts, although in that case the text system will usually approximate or substitute an oblique (slanted text) face if the font file doesn't contain a true italic typeface. If you really care about how your text looks, use fonts that have proper weighted versions (bold, light, and so on) and if used, proper italic versions. The calculated and approximated versions are never as nice looking.

Similarly, properties such as `FontStretch` are available to support stretching or condensing the text, but few fonts support actual stretching. In those cases, 100% normal size version is used instead, with no stretching.

The `TextBlock` is a regular framework element, participating in layout. You can change its horizontal and vertical alignment, margins, padding, and much more, just as you did with shapes. This provides you with significant flexibility when you use multiple `TextBlock` elements on the same page.

Shown in figure 8.1 is the simplest use of the `TextBlock`: Each block has a single bit of text, the entirety of which will have a single font style applied to it via the `FontSize`, `FontWeight`, `FontFamily`, and `Foreground` properties. But if you thought a single `TextBlock` was limited to displaying only a single font style, you'd be incorrect; there are other elements that can work with the `TextBlock` to provide additional capabilities. These elements are called *inlines*.

8.1.2 Inlines

An inline is an element that resides inside other text elements and provides most of the functionality of a full `TextBlock`. Unlike the `TextBlock`, a type derived from the `Inline` class can't be used by itself in XAML; it must reside in another text element. Typically, inlines are of the `Run` type but could also be of the `Span` or `LineBreak` type. The `TextBlock` itself can even contain a collection of inlines to display text using a variety of formatting options, simulating the functionality of rich text or HTML. For example, by combining inlines, you can create the awesome text shown in figure 8.2 using a single `TextBlock`.

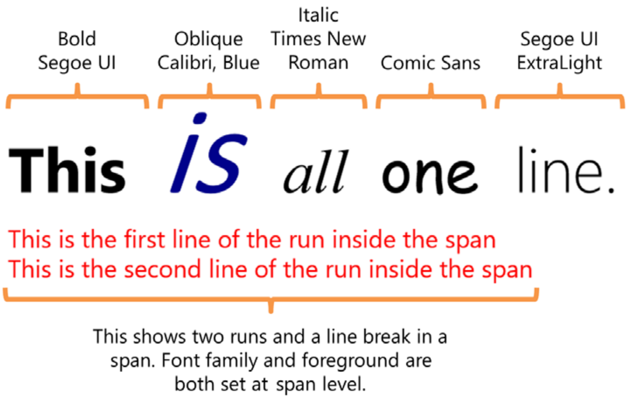

Bold
Segoe UI
Oblique
Calibri, Blue
Italic
Times New
Roman
Comic Sans
Segoe UI
ExtraLight
This *is* *all* one line.
This is the first line of the run inside the span
This is the second line of the run inside the span
This shows two runs and a line break in a span. Font family and foreground are both set at span level.

Figure 8.2 A single `TextBlock` with multiple inlines. Behold the awesome use of font faces and styles. Millions of designer voices cried out in terror and were suddenly silenced by the effectiveness of my Comic Sans placement.

In XAML, a `Run` is approximately equivalent to an HTML `span` tag. The XAML `Span`, somewhat confusingly, is an inline used to group other inlines; for example, a number of `Run` inlines that should all share the same formatting. And, of course, the `LineBreak` inline is equivalent to the HTML `br` tag. The next listing shows how to combine these inlines to create the image shown in figure 8.2.

Listing 8.2 A `TextBlock` with multiple inlines

```
<Grid Background="White">
  <TextBlock FontSize="75"
             Foreground="Black"
             Margin="50">
    <Run Text="This "
         FontFamily="Segoe UI"
         FontWeight="Bold"/>
```

Properties inherited by inlines

Runs all on one line

```
    <Run Text="is "
         FontSize="150"
         FontFamily="Calibri"
         FontStyle="Oblique"
         Foreground="DarkBlue"/>
    <Run Text="all "
         FontFamily="Times New Roman"
         FontStyle="Italic"/>
    <Run Text="one "
         FontFamily="Comic Sans MS" />
    <Run Text="line."
         FontWeight="ExtraLight"/>
    <LineBreak />
    <Span FontSize="30"
          Foreground="Red">
      <Run Text="This is the first line of the run inside the span" />
      <LineBreak />
      <Run Text="This is the second line of the run inside the span" />
    </Span>
  </TextBlock>
</Grid>
```

Runs all on one line

Line break

Properties inherited inside span

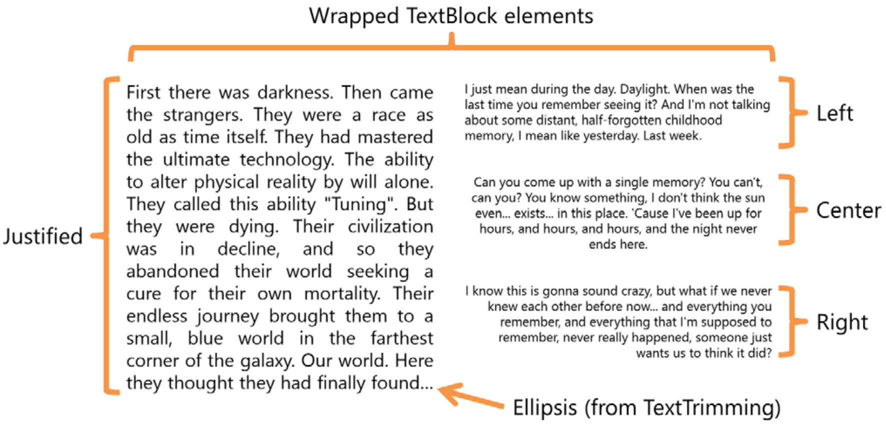
This listing shows how to create very rich text using a single `TextBlock`. For truly rich text, you'll likely want to use the `RichTextBlock` covered later in this chapter. But for a little emphasis here and there, this is just the thing. One thing to note is the inheritance of styles. Every run in the `TextBlock` inherits the base `TextBlock` settings unless they explicitly override them. Similarly, every inline in the `Span` inherits the `Span`'s inline settings (which may also be inherited from the `TextBlock`). This makes it easier to standardize the presentation while tweaking it in specific individual inlines.

> **NOTE** Additional types of inlines, supported but not commonly used with the `TextBlock`, are the `Underline`, `Italic`, and `Bold` inlines. These are specializations of `Span` that cause all included elements to be rendered with an underline, italic text, or bold text, respectively. More on those in the `RichTextBlock` section.

All of the examples in this section have so far used single lines of text, which fit conveniently on the display, but which weren't wrapped or otherwise aligned (or truncated) to the space they're displayed in. As you might expect, with the `TextBlock`, you have complete control over wrapping, truncation, and text alignment.

8.1.3 Wrapping, ellipsis, and alignment

Every word processor I've used since GeoWorks' GeoWrite on my Commodore 128 has included the capability to wrap text; to align text to the left, center, or right; and sometimes even justify it so it had even text edges on both sides (real-time justification was computationally expensive for those old 8-bit CPUs, so it was a luxury). XAML is no different in this respect in that any inline or `TextBlock` may use the `TextAlignment` property and `TextWrapping` properties to control alignment and word wrapping, respectively. In addition, XAML supports the ability to trim text that's too long for the



First there was darkness. Then came the strangers. They were a race as old as time itself. They had mastered the ultimate technology. The ability to alter physical reality by will alone. They called this ability "Tuning". But they were dying. Their civilization was in decline, and so they abandoned their world seeking a cure for their own mortality. Their endless journey brought them to a small, blue world in the farthest corner of the galaxy. Our world. Here they thought they had finally found...

I just mean during the day. Daylight. When was the last time you remember seeing it? And I'm not talking about some distant, half-forgotten childhood memory, I mean like yesterday. Last week.

Can you come up with a single memory? You can't, can you? You know something, I don't think the sun even... exists... in this place. 'Cause I've been up for hours, and hours, and hours, and the night never ends here.

I know this is gonna sound crazy, but what if we never knew each other before now... and everything you remember, and everything that I'm supposed to remember, never really happened, someone just wants us to think it did?

Figure 8.3 Text alignment, wrapping, and trimming in action

available space using the `TextTrimming` property, a very useful feature when displaying user-generated data of unknown length.

Figure 8.3 shows text alignment, wrapping, and text trimming in action. The first column shows a fully justified column of text. The text happens to be too long for the size of the `TextBlock` so it's trimmed with an ellipsis. In the second column, there are three paragraphs showing left, center, and right justification. Every paragraph here uses text wrapping.

To create this example, use the markup in the following listing.

Listing 8.3 Using the text alignment, wrapping, and trimming capabilities

```
<Grid Background="White">
  <Grid Margin="30">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock FontSize="36"
               TextTrimming="WordEllipsis"
               Margin="30"
               TextAlignment="Justify"
               TextWrapping="Wrap"
               Foreground="Black"
               Grid.Column="0">
      First there was darkness. Then came the strangers.
      They were a race as old as time itself. They had
      mastered the ultimate technology. The ability to
      alter physical reality by will alone. They called
      this ability "Tuning". But they were dying. Their
      civilization was in decline, and so they abandoned
      their world seeking a cure for their own mortality.
```

Full justification

Text wrapping

```
      Their endless journey brought them to a small, blue
      world in the farthest corner of the galaxy. Our
      world. Here they thought they had finally found
      what they had been searching for.
    </TextBlock>

    <StackPanel Grid.Column="1">
      <TextBlock FontSize="25"
                 Margin="30"
                 TextAlignment="Left"                   Left justified
                 TextWrapping="Wrap"                    Text wrapping
                 Foreground="Black">
        I just mean during the day. Daylight. When was the
        last time you remember seeing it? And I'm not
        talking about some distant, half-forgotten
        childhood memory, I mean like yesterday. Last
        week.
      </TextBlock>

      <TextBlock FontSize="25"
                 Margin="30"
                 TextAlignment="Center"                 Centered
                 TextWrapping="Wrap"                    Text wrapping
                 Foreground="Black">
         Can you come up with a single memory? You
         can't, can you? You know something, I don't
         think the sun even... exists... in this place.
         'Cause I've been up for hours, and hours, and
         hours, and the night never ends here.
       </TextBlock>

       <TextBlock FontSize="25"
                  Margin="30"
                  TextAlignment="Right"                 Right justified
                  TextWrapping="Wrap"                   Text wrapping
                  Foreground="Black">
        I know this is gonna sound crazy, but what
        if we never knew each other before now...
        and everything you remember, and everything
        that I'm supposed to remember, never really
        happened, someone just wants us to think it
        did?
      </TextBlock>
    </StackPanel>
  </Grid>
</Grid>
```

This listing creates the text shown in figure 8.3. Notice how the line breaks in the text in XAML don't correspond to the line breaks in the image. The XAML line break, just as in HTML, is counted just as whitespace unless you use the `xml:space="preserve"` attribute. I'll show you how to use that in the section on rich text.

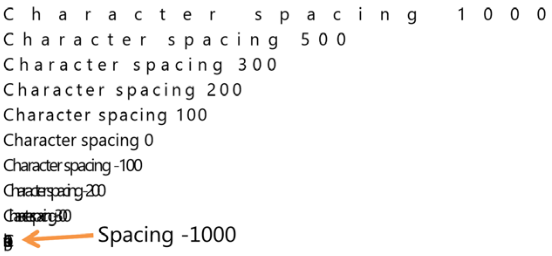
Wrapping is straightforward, as is justification. I'm going to assume you've used a word processor at some point in your life and can therefore figure those out. Text trimming is a little different in that it depends on the full size of the `TextBlock`, both width and height. The trimming occurs at the last word boundary, which leaves enough room for the ellipsis within the allocated space.

Full justification adds space between words in order to pad the text. There's another way you can increase the width of the same string of text: through character spacing.

8.1.4 Character spacing

Available on any `TextElement` (from which the elements `Block` and `Inline` are derived), `Control`, or `TextBlock`, the `CharacterSpacing` property controls the amount of space between individual characters.

The value for the `CharacterSpacing` property is expressed in thousandths of an em, or font size, with zero being the default. For example, if the `CharacterSpacing` is set to 200, the spacing will be 200/1000 * 1 em. The size of 1 em is itself defined by the font, but on average, the typical capital letter takes up about 0.7 em. Figure 8.4 shows the results of character spacing varying from 1000 to -1000, with the latter being squished and, with this font, mostly illegible.



Character spacing 1000
Character spacing 500
Character spacing 300
Character spacing 200
Character spacing 100
Character spacing 0
Character spacing -100
Character spacing -200
Character spacing -300

Figure 8.4 The effect of the `CharacterSpacing` property with values ranging from 1000 to -1000. Normal character spacing is 0. -1000 spacing is good to simulate a squashed spider.

If you went the full distance to -1000 as shown here, you would end up with every character stacked in a single spot. Although a hack, this can be useful if you have to overlay two characters to create a third.

The markup required to create this example is shown in the next listing.

Listing 8.4 Controlling character spacing

```
<Grid Background="White">
  <Grid.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontSize" Value="42" />
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Grid.Resources>

  <StackPanel Margin="10">
    <TextBlock CharacterSpacing="1000"
               Text="Character spacing 1000" />
    <TextBlock CharacterSpacing="500"
               Text="Character spacing 500" />
    <TextBlock CharacterSpacing="300"
               Text="Character spacing 300" />
    <TextBlock CharacterSpacing="200"
```

```
               Text="Character spacing 200" />
    <TextBlock CharacterSpacing="100"
               Text="Character spacing 100" />
    <TextBlock CharacterSpacing="0"
               Text="Character spacing 0" />
    <TextBlock CharacterSpacing="-100"
               Text="Character spacing -100" />
    <TextBlock CharacterSpacing="-200"
               Text="Character spacing -200" />
    <TextBlock CharacterSpacing="-300"
               Text="Character spacing -300" />
    <TextBlock CharacterSpacing="-1000"
               Text="Character spacing -1000" />
  </StackPanel>
</Grid>
```

In this listing, you have several `TextBlock` elements with the specified character spacing. Although `TextBlock` itself doesn't derive from `TextElement` (it derives directly from `FrameworkElement` for efficiency), note that you could also have used `Run`, `Paragraph`, `Span`, or any other element that derives from `TextElement`.

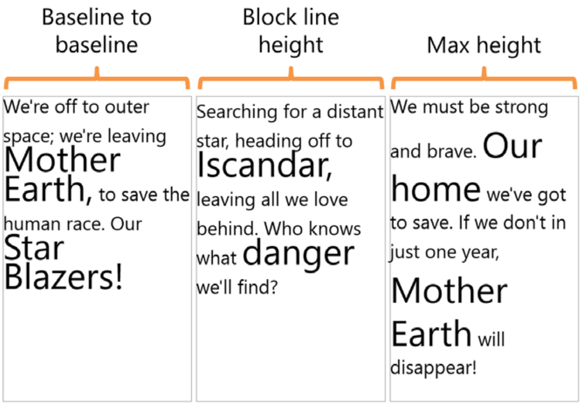
`CharacterSpacing` is all about horizontal space. While designers and typography aficionados will certainly use this, a more commonly used text layout property is `LineHeight`.

8.1.5 Line spacing

For readability (or for padding the length of your high school term paper), line spacing is key. In most word processors, you set the line spacing to one of a few standard spacing values. WinRT supports any arbitrary line height as well as three different line-stacking strategies. Figure 8.5 shows the three strategies all with the same 65 px line height.

You can control the spacing between lines of text in a single text element using the `LineHeight` property. XAML supports three types of line-stacking strategies, which come into play when there's more than a single font size on the line:

- *Baseline to baseline*—If specified, this adds the `LineHeight` value to the baseline of the previous line to calculate the correct spacing. If no `LineHeight` is specified,



We're off to outer space; we're leaving Mother Earth, to save the human race. Our Star Blazers!

Searching for a distant star, heading off to Iscandar, leaving all we love behind. Who knows what danger we'll find?

We must be strong and brave. Our home we've got to save. If we don't in just one year, Mother Earth will disappear!

Figure 8.5 Three `TextBlock` elements showing three different line-stacking strategies. A line height of 65 px was used in each case. Concentrate on the lines with more than one font size to see how the strategies are different. If you grew up watching *Star Blazers* in the United States in the '70s and '80s, the tune for this is surely rattling around in your head now.

it simply uses the default line height value. Characters on lines may overlap characters on other lines.

- *Block line height*—The stacking height is the height provided by `LineHeight`. Characters on lines may overlap characters on other lines. If no `LineHeight` is provided, each line is individually sized to hold its contents. In most cases, this will result in the same effect as baseline-to-baseline.
- *Max height*—The stacking height is the smallest height value that can contain every inline element on that line. If there's a `LineHeight` value provided, this is added to the calculated stacking height. Characters on lines won't overlap characters on other lines. Use this strategy when you want to guarantee that every line has the required amount of room.

The following listing shows how to create the image shown in figure 8.5 using the `LineHeight` and `LineStackingStrategy` properties.

Listing 8.5 Controlling line spacing

```
<Grid Background="White">
  <Grid.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontSize" Value="42" />
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Grid.Resources>
  <Grid Margin="40">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Rectangle Stroke="Gray" Grid.Column="0" Margin="5"/>
    <TextBlock LineHeight="65" TextWrapping="Wrap"
               Grid.Column="0" Margin="5"
               LineStackingStrategy="BaselineToBaseline">
      <Run>We're off to outer space; we're leaving </Run>
      <Run FontSize="80">Mother Earth, </Run>
      <Run>to save the human race. Our </Run>
      <Run FontSize="80">Star Blazers!</Run>
    </TextBlock>

    <Rectangle Stroke="Gray" Grid.Column="1" Margin="5" />
    <TextBlock LineHeight="65" TextWrapping="Wrap"
                 Grid.Column="1" Margin="5"
                 LineStackingStrategy="BlockLineHeight">
      <Run>Searching for a distant star, heading off to </Run>
      <Run FontSize="80">Iscandar, </Run>
      <Run>leaving all we love behind. Who knows what </Run>
      <Run FontSize="80">danger </Run>
      <Run>we'll find?</Run>
    </TextBlock>
```

BaselineToBaseline

BlockLineHeight

```
    <Rectangle Stroke="Gray" Grid.Column="2" Margin="5"/>
    <TextBlock LineHeight="65" TextWrapping="Wrap"
               Grid.Column="2" Margin="5"
               LineStackingStrategy="MaxHeight">
      <Run>We must be strong and brave. </Run>
      <Run FontSize="80">Our home</Run>
      <Run>we've got to save. </Run>
      <Run>If we don't in just one year, </Run>
      <Run FontSize="80">Mother Earth</Run>
      <Run>will disappear!</Run>
    </TextBlock>
  </Grid>
</Grid>
```

MaxHeight

This listing shows the interaction between `LineHeight` and the three different line-stacking strategies. The line height applies regardless of strategy, but the two properties, `LineHeight` and `LineStackingStrategy`, are typically used together.

As you develop apps, you're going to see and use a lot of `TextBlock` elements. Now that I've introduced you to them, I hope you also use inlines to augment the capabilities the `TextBlock` provides. In many cases, you'll find that the `TextBlock` provides a convenient way to group a number of related inlines to provide a level of cohesion you wouldn't otherwise get with a group of `TextBlock` elements.

The `TextBlock` also supports a number of essential text layout features, features shared by most `TextElement`-derived types. Wrapping of text is essential; it's really hard to create a fluid UI and support different screen resolutions, orientations, and snapped states without it. Text trimming also helps in this regard by making it possible to truncate text intelligently, ensuring it fits the available space. Having control over character spacing and line spacing provides you with more knobs you can tweak to help ensure that your text is both readable and beautiful.

In many cases, you'll need to display text that combines multiple fonts and styles and even inline UI elements like hyperlinks. Consider, for example, displaying a block of text from a social network. In such a case, you'll want something a little more capable than just a regular `TextBlock`; you'll want to use a `RichTextBlock`.

8.2 Rich and multicolumn text

I use the `TextBlock` more than anything else, but sometimes I need to do a little more than would be reasonable with the `TextBlock`. Maybe I need to incorporate some sort of UI element like a hyperlink in the middle of a run of text. This happens all the time when displaying messages from Twitter, Facebook, and other sources. I could hack something together with `StackPanel` elements, and maybe clever use of margins, but in the end, it would be hacky and brittle.

For times when the `TextBlock` just doesn't have enough flexibility, there's the `RichTextBlock`. This is a somewhat heavier element, but it includes functionality that you can't get in anything else.

The `RichTextBlock` is useful in two main ways:

- It provides a way to load text and UI elements together into the same flow.
- It supports multicolumn and linked text.

Notice that one thing left off that list is the ability to load actual rich text. *The `RichTextBlock` element doesn't include support for loading actual RTF (Rich Text Format) text.* If you need that capability, don't look at the `RichTextBlock`; you'll need the `RichEditBox`.

> **TIP** The `RichEditBox`, which provides editing capabilities and the ability to work with the Rich Text Format, is very similar to the `RichTextBlock` but with the notable addition of being able to work directly with RTF. The `RichTextBlock` itself can't directly work with RTF.

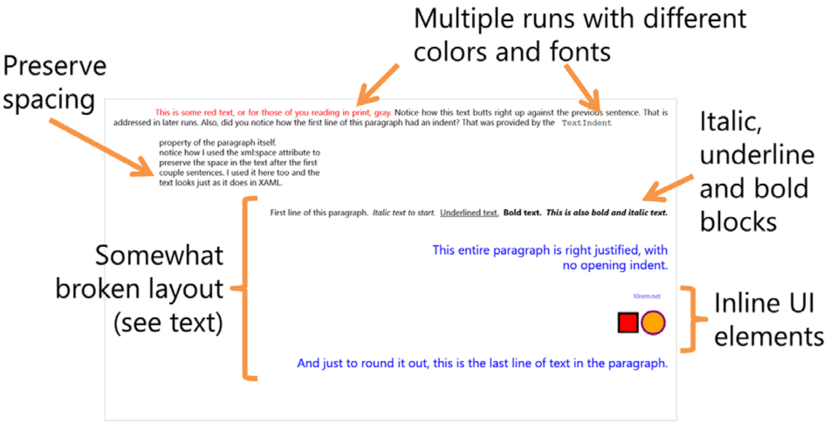
These capabilities, inline UI elements and multicolumn/linked text, may seem an incremental improvement over the `TextBlock` element, but they would add enough CPU and memory weight to the element to make it a good idea to factor the functionality into the `RichTextBlock` element.

In this section we'll take a look at how to use the `RichTextBlock` to display text containing multiple fonts, colors, and even XAML UI elements. Then, we'll jump into one of my favorite uses of the `RichTextBlock`: multicolumn and linked text.

8.2.1 Rich text

The `RichTextBlock` is the main way to display read-only rich text in XAML apps. It's similar to a `TextBlock` but adds support for paragraphs and inline UI elements within the control. As mentioned earlier, the `RichTextBlock` can't work directly with RTF; you'll need the `RichEditBox` for that.

What the `RichTextBlock` can do well is support combining different types of content in the same element. Unlike the `TextBlock`, it doesn't use inlines but instead uses items that inherit from `Block`. `Block` is conceptually similar to `Inline` and related by its common inheritance from `TextElement` but is optimized for use with rich text and UI elements. A `Block` has support for the line-stacking and text-alignment properties


Multiple runs with different colors and fonts
Preserve spacing
Italic, underline and bold blocks
Somewhat broken layout (see text)
Inline UI elements
10rem.net

This is some red text, or for those of you reading in print, gray. Notice how this text butts right up against the previous sentence. That is addressed in later runs. Also, did you notice how the first line of this paragraph had an indent? That was provided by the `TextIndent`

property of the paragraph itself.
notice how I used the xml:space attribute to
preserve the space in the text after the first
couple sentences. I used it here too and the
text looks just as it does in XAML.

First line of this paragraph. *Italic text to start.* Underlined text. **Bold text.** ***This is also bold and italic text.***

This entire paragraph is right justified, with
no opening indent.

And just to round it out, this is the last line of text in the paragraph.

Figure 8.6 A `RichTextBlock` with paragraphs and UI elements. Notice the mixture of fonts and colors, the inclusion of two vector graphics shapes, and a `HyperlinkButton` control. See the sidebar "A RichTextBlock layout bug with xml:space" for why I have some of the text labeled as "somewhat broken layout." That section inside the bracket would normally appear as one continuous set of lines.

previously covered; so in some ways, it's almost a mini `TextBlock` itself. In addition, however, it adds the ability to include arbitrary UI elements like hyperlinks, buttons, and more.

Figure 8.6 shows a single `RichTextBlock` taking up the entire page. Inside that `Block`, there are numerous examples of different fonts and colors, text formatting and positioning, inline UI elements and more.

The following listing shows how to create the `RichTextBlock` in figure 8.6, including loading a `StackPanel` with two vector graphics shapes right into the `RichTextBlock`.

Listing 8.6 Creating the `RichTextBlock`

```
<Grid Background="White">
  <RichTextBlock Foreground="Black" FontSize="20" Margin="20">
    <Paragraph TextIndent="100" TextAlignment="Justify">
      <Run Foreground="Red">
        This is some red text, or for those of you reading in print, gray.
      </Run>
      <Run>
        Notice how this text butts right up against the
        previous sentence. That is addressed in later runs.
        Also, did you notice how the first line of this
        paragraph had an indent? That was provided by the
      </Run>
      <Run xml:space="preserve" FontFamily="Courier New"
           Text=" TextIndent " />
      <Run xml:space="preserve">
        property of the paragraph itself.
        notice how I used the xml:space attribute to
        preserve the space in the text after the first
        couple sentences. I used it here too and the
        text looks just as it does in XAML.
      </Run>
    </Paragraph>

    <Paragraph TextAlignment="Right" TextIndent="0">
      <Run>First line of this paragraph. </Run>
      <Italic xml:space="preserve"> Italic text to start. </Italic>
      <Underline xml:space="preserve">Underlined text.</Underline>
      <Bold xml:space="preserve"> Bold text. </Bold>
      <Run xml:space="preserve"
           FontWeight="Bold" FontStyle="Italic"
           Text="This is also bold and italic text. " />
      <Span Foreground="Blue" FontSize="30" >
        <Run>
          This entire paragraph is right justified, with
          no opening indent.
        </Run>
        <InlineUIContainer>
          <HyperlinkButton Content="10rem.net"
                           NavigateUri="http://10rem.net" />
        </InlineUIContainer>
        <InlineUIContainer>
```

Paragraph with indent

Red text

xml:space instance

Second paragraph, no indent

Span with font size

HyperlinkButton in UI container

StackPanel in UI container

```
          <StackPanel Orientation="Horizontal">
            <Rectangle Width="50" Height="50" Fill="Red"
                       Stroke="Black" StrokeThickness="5" />
            <Ellipse Width="60" Height="60" Fill="Orange"
                     Stroke="Purple" StrokeThickness="5"
                     Margin="5"/>
          </StackPanel>
        </InlineUIContainer>
        <Run>
          And just to round it out, this is the last line of
          text in the paragraph.
        </Run>
      </Span>
    </Paragraph>
  </RichTextBlock>
</Grid>
```

This listing shows all the major features of the `RichTextBlock` and the `Block` and `Span` elements it can contain. The `RichTextBlock` itself can contain only `Block`-derived elements, typically `Paragraph`s. A `Paragraph` can, however, contain all the same inlines that a `TextBlock` can contain. Therefore, everything you learned earlier in this chapter can be applied here. In addition, the `Paragraph` can also hold a new element called the `InlineUIContainer`.

The `InlineUIContainer` enables you to put XAML controls and elements inside the `RichTextBlock` text flow. Normally, this would be used to add in hyperlinks and images, but it could be really anything. Each `InlineUIContainer` can contain only a single element. If you have more than one element, and you want the elements to flow with the rest of the text, put them in different containers. If you want to keep a set of elements grouped in a specific visual layout, put them in a `Grid`, `StackPanel`, or other panel, and make that panel the child of the `InlineUIContainer`.

A RichTextBlock layout bug with xml:space

I've been asked many times how to preserve carriage returns and, especially, spaces in XAML. This is something that can frustrate designers looking to get a bunch of hard-coded text into a UI.

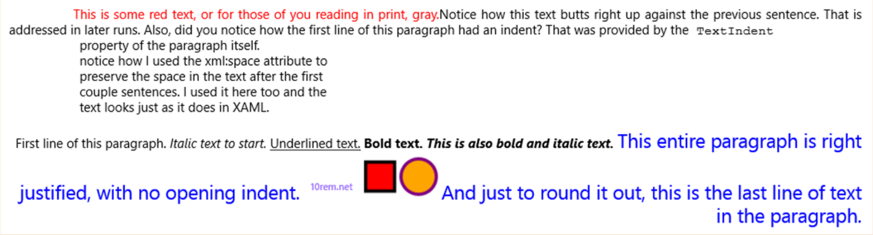
Notice how the text had the in-XAML whitespace preserved in the output. You can see that in figure 8.6 and, assuming editing this book for print hasn't changed any of the spacing, listing 8.6. This is handled through the use of the `xml:space` attribute. Use this attribute sparingly, though.

When writing this chapter, I uncovered a bug in the runtime layout process: The design surface looks correct, but the use of `xml:space` anywhere in the `Paragraph` causes the layout of the remaining blocks, especially `InlineUIContainer` elements, to be laid out incorrectly at runtime. Essentially what happens is that if the preserve appears anywhere in the paragraph, it applies to everything in the paragraph.

The VS2012 designer, which is based on Expression Blend, shows the expected runtime representation. For this release of WinRT, though, it doesn't quite match reality.

(continued)

Here's the designer view for the same `RichTextBlock` layout we've worked with in this chapter.



This is some red text, or for those of you reading in print, gray.Notice how this text butts right up against the previous sentence. That is addressed in later runs. Also, did you notice how the first line of this paragraph had an indent? That was provided by the `TextIndent`
property of the paragraph itself.
notice how I used the xml:space attribute to
preserve the space in the text after the first
couple sentences. I used it here too and the
text looks just as it does in XAML.

First line of this paragraph. *Italic text to start.* Underlined text. **Bold text.** ***This is also bold and italic text.*** This entire paragraph is right justified, with no opening indent. 10rem.net 🟥 ● And just to round it out, this is the last line of text in the paragraph.

You'll notice that in the figure, the entire last paragraph sticks together properly and flows left to right, including the UI elements.

Because you may run into this yourself, I thought it better to show you the version with bug and not simply work around it. When you try this example, run it as shown, and then remove every last instance of the `xml:space` attribute to see the difference in layout. That's the workaround as well: Simply avoid using that property when you need to mix and match pre-laid-out text with fluid layout.

As the website error pages always say, "The appropriate people have been notified," except I actually did. :)

The `RichTextBlock` builds on the trail blazed by the `TextBlock` and adds essential features that make it easier to work with mixed blocks of text. In addition, you can use the `RichTextBlock` to hold other UI elements, making it almost like a whole new type of panel.

This element is useful for more than just rich text with inline UI elements. One other great feature is the ability to link one or more `RichTextBlockOverflow` elements to a single `RichTextBlock` to allow the text to flow through the UI.

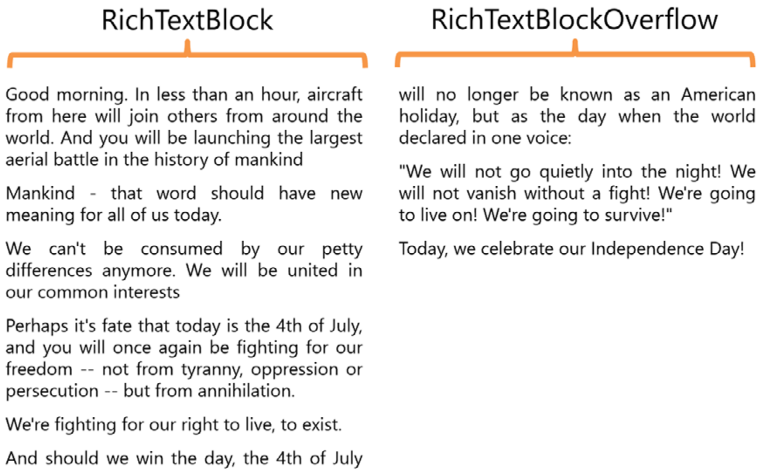
8.2.2 Multicolumn and linked text

One feature that has been requested in HTML and XAML since the beginning of time[2] (WPF flow documents notwithstanding) is the ability to have linked and multicolumn text. This enables magazine- and newspaper-style layouts and gives designers a lot more flexibility with how to lay out the content in applications.

In XAML, multicolumn and freeform linked text both build on the `RichTextBlock` element. The easiest way to use linked text is to have multiple columns. Fortunately, this is also one of the most commonly requested text scenarios for XAML and for web apps.

---

[2] Yes, the beginning of time. The request for multicolumn text in HTML originally appeared on a cave wall in southern France, apparently painted during the time the HTML2 spec was being drafted.



Good morning. In less than an hour, aircraft from here will join others from around the world. And you will be launching the largest aerial battle in the history of mankind

Mankind - that word should have new meaning for all of us today.

We can't be consumed by our petty differences anymore. We will be united in our common interests

Perhaps it's fate that today is the 4th of July, and you will once again be fighting for our freedom -- not from tyranny, oppression or persecution -- but from annihilation.

We're fighting for our right to live, to exist.

And should we win the day, the 4th of July will no longer be known as an American holiday, but as the day when the world declared in one voice:

"We will not go quietly into the night! We will not vanish without a fight! We're going to live on! We're going to survive!"

Today, we celebrate our Independence Day!

Figure 8.7 Two columns created using a single `RichTextBlock` and a single `RichTextBlockOverflow` element

The implementation in WinRT XAML doesn't limit you to an arbitrary number of columns. Instead, a `RichTextBlock` may be linked to a single `RichTextBlockOverflow` element.

In figure 8.7 you can see two columns of text displaying the famous and somewhat over-the-top speech from the cult classic movie *Independence Day*. The text is too large for the `RichTextBlock` so it freely flows from the first column to the second should you resize the screen or otherwise change the size of the first column. You can even select text starting in the first column and finishing in the second, as though it's one giant, selectable text element.

The following listing shows the markup required to create the two-column layout with text overflow.

Listing 8.7 Multicolumn text using a `RichTextBlock`

```
<Grid Background="White">
  <Grid Margin="30">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <RichTextBlock Grid.Column="0" Margin="20"
                   Foreground="Black" TextAlignment="Justify"
                   FontSize="30"
     OverflowContentTarget="{Binding ElementName=SecondColumnText}">
      <Paragraph Margin="10,20,10,20">
        Good morning. In less than an hour, aircraft from here will
        join others from around the world. And you will be launching
```

Two equal columns

Justify text

Link to overflow

```
        the largest aerial battle in the history of mankind
      </Paragraph>
      <Paragraph Margin="10,20,10,20">
        Mankind - that word should have new meaning for all of us today.
      </Paragraph>
      <Paragraph Margin="10,20,10,20">
        We can't be consumed by our petty differences anymore. We
        will be united in our common interests
      </Paragraph>
      <Paragraph Margin="10,20,10,20">
        Perhaps it's fate that today is the 4th of July, and you will
        once again be fighting for our freedom -- not from tyranny,
        oppression or persecution -- but from annihilation.
      </Paragraph>
      <Paragraph Margin="10,20,10,20">
        We're fighting for our right to live, to exist.
      </Paragraph>
      <Paragraph Margin="10,20,10,20">
        And should we win the day, the 4th of July will no longer be
        known as an American holiday, but as the day when the world
        declared in one voice:
      </Paragraph>
      <Paragraph Margin="10,20,10,20">
        "We will not go quietly into the night! We will not vanish
        without a fight! We're going to live on! We're going to survive!"
      </Paragraph>
      <Paragraph Margin="10,20,10,20">
        Today, we celebrate our Independence Day!
      </Paragraph>
    </RichTextBlock>

    <RichTextBlockOverflow x:Name="SecondColumnText"                 Overflow
                           Grid.Column="1"
                           Margin="20"/>
  </Grid>
</Grid>
```

The two things to focus on are the `OverflowContentTarget` property of the `RichTextBlock` and the name assigned to the `RichTextBlockOverflow`. Notice how I use element binding (covered in chapter 10) to link the `RichTextBlock` to the `RichTextBlockOverflow`. I simply had to make sure that the overflow had an `x:Name` and then I could use that in the binding statement in the `RichTextBlock`. This is the standard way to link the two elements together in markup. You could also do this in code, but it's easier to do here.

One final detail to call out: The overflow has no content or styling of its own, other than the margin. Everything else—the font face and color and the actual text—comes from the `RichTextBlock` element.

I would be remiss if I talked about beautiful text and rich text but didn't cover one thing that makes it possible to go above and beyond the plain: OpenType. XAML has awesome OpenType feature support, as you'll see in the next section.

8.3 OpenType text

Text formatting and rendering options aside, typically what you see with a typeface is what you get. There are no provisions for special relationships between characters or embellishments around words. With the emphasis on beautiful and easily understood text in Windows 8, it's now more important than ever to go that extra step to ensure your text is rendering as well as possible and is both beautiful and easily understood.

OpenType fonts have support for multiple ways to render characters or groups of characters. They have context-aware glyphs that may be used when certain characters are positioned next to each other or when numbers are positioned in specific ways relative to other characters. They also include stylistic sets that are based on the same core font but that offer embellishments that you'd normally expect to find only in another entirely new font file.

All of these things combine to make it easier for you to take the extra step to provide truly beautiful and compelling text, better support for the brand you're promoting, and generally make sure the text is easier for the user to read and understand.

In this section, we'll take a look at some of the supported OpenType features including ligatures, alternatives and stylistic sets, font capitals, fractions and number formats, and variants including superscript and subscript.

8.3.1 Ligatures

A ligature is one or more characters joined together as a single glyph. Typically, this is done with two characters linked together based on context. In traditional printing, the ligature would be a single printing block. In computer typography, the font defines how the ligature is handled. Not only does this look more refined, but it often increases readability.

The most common ligatures revolve around the letters following the letter *f*. Figure 8.8 shows both the no-ligature version and the version using OpenType ligatures with the Gabriola font.

The use of ligatures removes the awkward splicing of the crossbar on the lowercase *f* and *t* characters (a combination I see often when typing out my employer's name, "Microsoft"). It also removes the dots on the lowercase *i* and *j* characters, because those mash up against the *f*. Finally, you'll notice that the top of the *f* is handled differently when followed by other tall characters such as *f* and *l*.

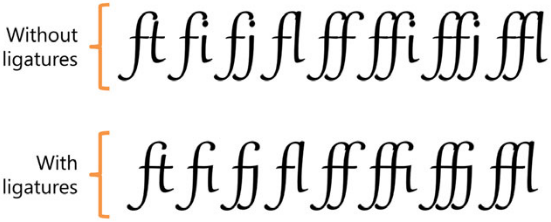

Without ligatures
ft fi fj fl ff ffi ffj ffl
With ligatures
ft fi fj fl ff ffi ffj ffl

Figure 8.8 Standard ligatures in the OpenType Gabriola font. Pay particular attention to the dots on the *i* and *j* characters, the crossbars in the *t* and *f*, and the tops of the lowercase *f* characters and how they connect to the next character.

If the font supports it, Windows enables or disables the use of standard ligatures such as those shown here through the `Typography.StandardLigatures` attached property. The next listing shows the markup required to set this property.

Listing 8.8 Using ligatures

```
<Grid Background="White">
  <StackPanel Margin="30">
    <TextBlock Text="ft fi fj fl ff ffi ffj ffl"
               Foreground="Black"
               HorizontalAlignment="Center"
               FontSize="200"
               FontFamily="Gabriola"
               Typography.StandardLigatures="False" />      ◁ Disabled
    <TextBlock Text="ft fi fj fl ff ffi ffj ffl"
               Foreground="Black"
               HorizontalAlignment="Center"
               FontSize="200"
               FontFamily="Gabriola"
               Typography.StandardLigatures="True" />       ◁ Enabled
  </StackPanel>
</Grid>
```

In addition, if the OpenType font supports them, WinRT XAML enables contextual ligatures, discretionary ligatures, and historical ligatures through the `ContextualLigatures`, `DiscretionaryLigatures`, and `HistoricalLigatures` properties, respectively. You use these the same way you use the `StandardLigatures` property.

Historical ligatures are ones that were once standard but are no longer commonly used. If you're looking to make your app appear classical (or maybe steampunk or a love letter from your favorite pirate), and the font supports them, historical ligatures can add real character.

Contextual ligatures are ones that the font designer believes are appropriate for use with the font. Enabling both standard and contextual ligatures will give you the complete set the font designer felt were appropriate for normal use.

Discretionary ligatures are ones that the font designer included for specific situations and that may not apply to general use throughout the entire body of your text.

Ligatures can help increase the readability and aesthetics of your text. Another way to really fancy things up is to use contextual alternates and stylistic sets.

8.3.2 Stylistic sets

Stylistic sets are alternate representations of glyphs in a font. These can range from very subtle variations of characters, suitable for more formal documents, all the way up to very fancy and fantastic renderings of the loops, as well as decorative elements. It's up to the designer to create these in such a way as to stay consistent with the rest of the font but provide interesting variation.

A font designer may include up to 20 optional stylistic sets in a font, and each stylistic set may include any subset of the characters of the font. The Gabriola font we're using here includes seven stylistic sets.

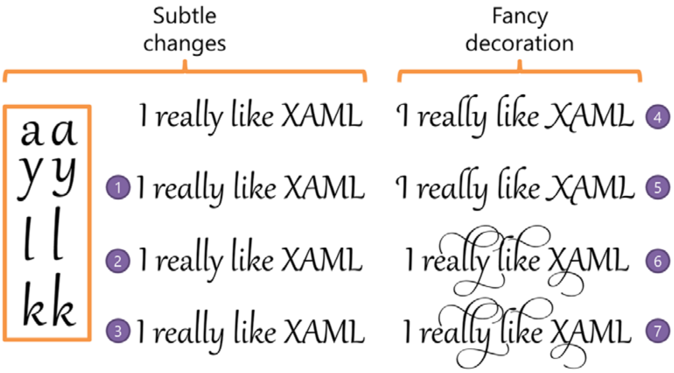

Subtle changes
Fancy decoration
a a
y y
l l
k k
I really like XAML
① I really like XAML
② I really like XAML
③ I really like XAML
I really like XAML ④
I really like XAML ⑤
I really like XAML ⑥
I really like XAML ⑦

Figure 8.9 Stylistic sets for the Gabriola font as rendered by Windows. Note not only the obvious changes in the second column but also the subtle treatment given to the *a*, *y*, *l*, and *k* characters in the first column (called out in the block to the left).

Figure 8.9 shows all seven Gabriola stylistic sets displayed in WinRT XAML.

The sets get progressively fancier, with sets 6 and 7 showing real flare. Even sets 4 and 5 start to get going with the treatment of the lowercase *l* and the capital *A*. I particularly like the treatment given to the lowercase *y* character.

The following listing shows how to enable the different stylistic sets from markup.

Listing 8.9 A number of stylistic sets for the Gabriola font

```
<Grid Background="White">
  <Grid Margin="30">
    <Grid.Resources>
      <Style TargetType="TextBlock">
        <Setter Property="Text" Value="I really like XAML" />
        <Setter Property="FontFamily" Value="Gabriola" />
        <Setter Property="Foreground" Value="Black" />
        <Setter Property="FontSize" Value="100" />
        <Setter Property="HorizontalAlignment" Value="Center" />
      </Style>
    </Grid.Resources>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0" />

    <TextBlock Grid.Row="1" Grid.Column="0"
               Typography.StylisticSet1="True"/>
    <TextBlock Grid.Row="2" Grid.Column="0"
               Typography.StylisticSet2="True" />
    <TextBlock Grid.Row="3" Grid.Column="0"
               Typography.StylisticSet3="True" />
```

Implicit style

Normal style

First column stylistic sets

```
    <TextBlock Grid.Row="0" Grid.Column="1"
               Typography.StylisticSet4="True" />
    <TextBlock Grid.Row="1" Grid.Column="1"
               Typography.StylisticSet5="True" />
    <TextBlock Grid.Row="2" Grid.Column="1"
               Typography.StylisticSet6="True" />
    <TextBlock Grid.Row="3" Grid.Column="1"
               Typography.StylisticSet7="True" />
    </Grid>
</Grid>
```

Second column stylistic sets

In this example, I used an implicit style to avoid repeating the same properties for every `TextBlock`. In this example, I used an implicit style (chapter 7) to avoid repeating the same properties for every `TextBlock`. Think of it as simply repeating the declared values for each and every `TextBlock` on the page, so they all have the same `Text`, `FontFamily`, `FontSize`, and `HorizontalAlignment` properties.

Selecting the stylistic set is as simple as setting its numbered `StylisticSetN` property to `true`. If there's no set in that spot, you'll receive the default font. If you want to check for support for alternates, you can open up Microsoft Word, right-click the text, and select the advanced font properties. You'll be able to easily browse many of the OpenType features of the font, including stylistic sets. Set only one stylistic set property to `true` at a time.

While not required, the `ContextualAlternates` property is often used with stylistic sets in order to fine-tune the relationships between the characters. The effects of setting this property to `true` are typically more subtle than a stylistic set. Alternates are glyphs that can be substituted for a standard glyph. Contextual alternates are ones that are automatically substituted based on the context of the original glyph. For example, picture the relationship between characters in a hand-written cursive typeface—characters following a lowercase *o* often start differently than ones following, say, a lowercase *l*.

Looking for other OpenType fonts?

Gabriola happens to be a personal favorite, but you can also find several other fonts included with the Windows SDK and the Sample OpenType Font Pack. This pack includes Kootenay, Lindsey, Miramonte, Miramonte Bold, Pericles, Pericles Light, Pescadero, and Pescadero Bold. Each of these fonts implements various OpenType features.

8.3.3 *Font capitals*

Another way to modify the appearance of text is to convert it to all caps. You could do this simply by using ALL CAPS when you type the text, but many fonts include alternate representations for small caps, petite caps, titling case, and others. In WinRT XAML, this is supported by the `Typography.Capitals` attached property.

XAML supports the following optional types of capitals via the `Capitals` property: Normal, All Petite Caps, All Small Caps, Petite Caps, Small Caps, Titling, and Unicase.

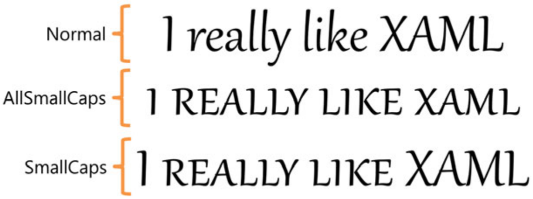


Figure 8.10
A comparison among normal text with capital letters and the `AllSmallCaps` and `SmallCaps` OpenType settings

The appearance of the type is dependent on how the font designer created it and could vary significantly from one font to the next. Figure 8.10 shows how `Normal`, `AllSmallCaps`, and `SmallCaps` appear in Gabriola.

You can see that the word *XAML* is smaller in the `AllSmallCaps` version, as expected. In addition, there are some slight differences in spacing, like between the characters in *XAML*. It's not necessarily the same as an algorithmic resizing of the letters; the designer has control over how the smaller letters appear.

The next listing shows how to use `AllSmallCaps`, `SmallCaps`, and `Normal` to create the text shown in figure 8.10.

Listing 8.10 Display text as all caps using OpenType

```
<Grid Background="White">
    <StackPanel Margin="30">
        <StackPanel.Resources>
            <Style TargetType="TextBlock">
                <Setter Property="Text" Value="I really like XAML" />
                <Setter Property="FontFamily" Value="Gabriola" />
                <Setter Property="Foreground" Value="Black" />
                <Setter Property="FontSize" Value="150" />
                <Setter Property="HorizontalAlignment" Value="Center" />
            </Style>
        </StackPanel.Resources>

        <TextBlock Typography.Capitals="Normal"/>
        <TextBlock Typography.Capitals="AllSmallCaps" />
        <TextBlock Typography.Capitals="SmallCaps" />
    </StackPanel>
</Grid>
```

Normal

All small caps

Small caps

As in the previous listing, I used the same resource block to eliminate repetition throughout the listing. The key thing to notice here is the `Typography.Capitals` attached property on the `TextBlock`.

8.3.4 Fractions and numbers

Numbers, and in particular fractions, present unique challenges when formatting text for display. Numbers have different alignment needs than plain text, especially when displaying a table of numbers to be added together. Numbers in fractions also need to be displayed differently when they're the numerator or denominator. Even that changes if you consider side-by-side fractions versus the traditional top-over-bottom fractions.

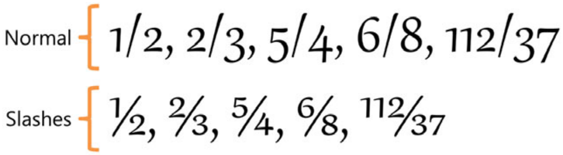


Figure 8.11
Normal and side-by-side slashed representations of fractions. I certainly prefer the slashes over the normal, myself.

XAML supports the different number alignment values through the `NumeralAlignment` and `NumeralStyle` attached properties of the `Typography` class. It even supports turning slashed zeros on or off using the `SlashedZero` attached property. Gabriola doesn't do anything special in those instances, so we'll leave them alone for this example.

If the OpenType font supports different fraction representations, so does XAML. Fortunately, Gabriola also supports two of the three fractional representations, as shown in figure 8.11.

Unlike say, typing a lengthy document in Microsoft Word[3] and having autosubstitution of fractions happen as you type, in XAML, you still have separate characters for each part of the fraction and aren't limited to the basic less-than-one fractions you normally get with substitution.

When formatting, XAML and OpenType automatically figure out the numerator and denominator based on the position of the slash character. That really keeps your markup clean. The following listing shows how to use this feature.

Listing 8.11 Representing fractions using OpenType

```
<Grid Background="White">
  <StackPanel Margin="30">
    <StackPanel.Resources>
      <Style TargetType="TextBlock">
        <Setter Property="FontFamily" Value="Gabriola" />
        <Setter Property="Foreground" Value="Black" />
        <Setter Property="FontSize" Value="175" />
        <Setter Property="HorizontalAlignment" Value="Left" />
      </Style>
    </StackPanel.Resources>

    <TextBlock Text="1/2, 2/3, 5/4, 6/8, 112/37"
               Typography.Fraction="Normal" />

    <TextBlock Text="1/2, 2/3, 5/4, 6/8, 112/37"
               Typography.Fraction="Slashed" />

    </StackPanel>
</Grid>
```

Normal

Slashes

In this listing, I simplified the style and removed the `Text` property setter, because I wanted slightly different text in each `TextBlock`. The markup here creates the text shown in figure 8.11. Note how I didn't need to specify numerator, denominator, or even superscript and subscript to make the numbers line up. That was all done automatically.

---

[3] I wouldn't know anything about typing lengthy documents in Microsoft Word, especially not phonebook-sized technical books.

8.3.5 Variants, superscript, and subscript

Sometimes fractions are approximated using superscript and subscript text. That's not particularly accurate and can actually be a real pain to do in some cases. It's much better to let WinRT handle that for you automatically.

But there are certainly times when you want to have control over superscript and subscript text. Footnotes and chemistry are two places where superscript and subscript prevail.

Superscript and subscript are supported by the `Typography.Variants` attached property. Those aren't the only two variants supported, however. Some fonts, Eastern fonts in particular, include support for several other variants including Inferior, Ordinal, and Ruby. Of course, we also have Normal for plain-old text.

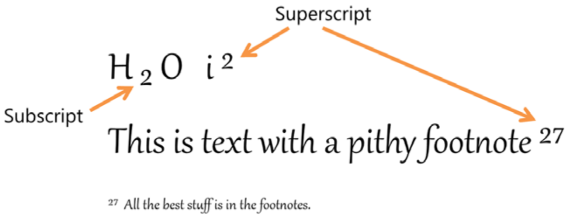

Superscript
Subscript
H₂O i²
This is text with a pithy footnote 27
27 All the best stuff is in the footnotes.

Figure 8.12 This shows the use of superscript and subscript in text using the `Typography.Variants` attached property.

Figure 8.12 shows both superscript and subscript as well as normal. Gabriola doesn't support any of the other variants.

Unlike fractions where the sizing is automatic, when using variants, you need to explicitly set the `Typography.Variants` property for each run of text you want affected. The next listing shows the markup required for this example.

Listing 8.12 Superscript and subscript variants

```
<Grid Background="White">
  <StackPanel Margin="30">
    <StackPanel.Resources>
      <Style TargetType="TextBlock">
        <Setter Property="FontFamily" Value="Gabriola" />
        <Setter Property="Foreground" Value="Black" />
        <Setter Property="FontSize" Value="100" />
        <Setter Property="HorizontalAlignment" Value="Left" />
      </Style>
    </StackPanel.Resources>

    <StackPanel Orientation="Horizontal">
      <TextBlock>
        <Run Text="H" Typography.Variants="Normal" />
        <Run Text="2" Typography.Variants="Subscript" />
        <Run Text="O" Typography.Variants="Normal" />
      </TextBlock>
      <TextBlock Margin="50 0 0 0">
```

Subscript

```
        <Run Text="i" Typography.Variants="Normal" />
        <Run Text="2" Typography.Variants="Superscript" />
      </TextBlock>
    </StackPanel>

    <TextBlock>
      <Run Text="This is text with a pithy footnote"
           Typography.Variants="Normal" />
      <Run Text="27"
           Typography.Variants="Superscript" />
    </TextBlock>

    <TextBlock FontSize="40">
      <Run Text="27 "
           Typography.Variants="Superscript" />
      <Run Text="All the best stuff is in the footnotes."
           Typography.Variants="Normal" />
    </TextBlock>
  </StackPanel>
</Grid>
```

Superscript

As in the other examples, I used a style to keep the listing short. I did override the font size in the `TextBlock` that represents a footnote, however. Another cool feature of styles.

This example had a good bit more markup because, as mentioned, I had to set the `Normal`, `Superscript`, or `Subscript` variant for each block of text. Also note how these properties are available on the `Run` element inside a `TextBlock` as well. They're portable to just about anything using text.

OpenType opens up a whole new world of typography options from the mundane to the super-fancy. WinRT's support for OpenType is excellent, supporting all the major features that a developer or designer could ask for. While they won't be used in every application, you'll find yourself able to use many of the features, especially variants and fractions, in business applications and more design-oriented applications alike.

The text system always renders text using the specified font, a specified fallback, or a default fallback if unspecified or if the font is unavailable. If the font you're using in your `TextBlock` or other control isn't a standard font, you may want to consider embedding it with your application.

8.4 Embedding fonts

The majority of Windows applications, at least initially, will use the standard Microsoft Segoe font family for displaying and editing text. Eventually, that will get as boring to look at as the old MS Sans Serif in desktop apps. There will come a time when you'll want to use a special font in your application. Maybe it's the typeface in use in your company logo, and you want to match it for consistent branding. Sometimes it's a slick headline font. Or perhaps it's just a sharp and readable font you want to use in your news-reading application. You could just use a nonstandard font and pray that it's already installed on an end user's machine, but trust me: That won't end well.

What do you do when you can't guarantee that end users will have that font on their machines? One way to tackle this problem is to embed the font into the application.

I'm not a lawyer

But that's not going to stop me from giving pseudo-legal advice (advice that you should very clearly understand is not from a lawyer). Before you go and embed that font, check its license. Most fonts don't legally allow embedding in applications. In fact, most fonts haven't even caught up with the idea that fonts can be used outside of documents.

Once the font foundries get out of the '80s and start allowing font embedding in applications, UIs will really start to shine.

In the meantime, I suggest you consult someone with a real legal background before embedding that font in your application. Resist the urge to use a font just because it's on your developer machine, installed with some other app like Microsoft Word or Adobe Photoshop. Instead, get explicit embedding rights or licenses. Seriously, I'm not a lawyer, and I don't even play one on TV.

Although there's no special tooling for it, WinRT apps support embedded fonts in applications. Simply add the font to your project and mark it as content. You can then refer to it by name using the format `FileName#FontName`:

```
<TextBlock FontFamily="/Assets/JosefinSansStd-Light.otf#Josefin Sans Std"
           FontSize="40"
           Text="This text uses an embedded font" />
```

The folder name */Assets* is the location where the original OTF (Open Type Font) or TTF (True Type Font) file is placed in the project. The name JosefinSansStd-Light is the name of the font file, and *Josefin Sans Std* is the name of the actual font. You can identify the font name by double-clicking the file and examining it in the font viewer.

Once you have fonts embedded in your application, they can be used anywhere you'd use a regular typeface. For example, they may be used in text boxes for gathering text, something we'll look at in the next chapter.

8.5 Summary

I don't consider myself a typography nerd, but I do love beautiful text. I tend to notice when ads on TV (especially now with HDTV available for just about every channel) take the effort to provide beautiful typography as part of the layout. I've always noticed the same in movies, and I'm starting to see some of the same things creep into app development.

WinRT XAML has such great support for typography because typography is so important to Windows 8. You only need look at the history of typography in Silverlight and WPF to realize that the emphasis on beautiful type was not always top on the feature list for the platforms. With WinRT, this changes. Not only is the type rendered more cleanly and more beautifully, but feature support goes above and beyond that required to stick a field label on a form.

The `TextBlock` is the most basic and most commonly used text element. It has support for the usual font properties, plus alignment and more. It also supports the concept of an inline, something that lets you combine many elements into a single `TextBlock`, with each element having its own formatting. You can wrap text, align it, and even truncate the text to fit the available display space.

The `RichTextBlock` builds on the `TextBlock` and can even use the same inlines. It also adds in the ability to select text and to have multicolumn text. One really cool thing about it is the support for embedding controls directly into the `RichTextBlock` itself. If you're an industrious XAML developer, I'll bet you could even use a `RichTextBlock` as a very flexible `Panel` substitute. Being somewhat lighter weight, the `TextBlock` is still the go-to element when you have a ton of them to put on a single page, but the `RichTextBlock` offers features that make it a compelling substitute.

Both the `RichTextBlock` and the `TextBlock` make use of OpenType fonts. Of course, you can set the font sizes, families, weight, and styles, but the `Typography` namespace enables you to dig more deeply into the features exposed by the different OpenType fonts. This helps ensure you're displaying them at their very best in every possible situation.

Finally, all that effort you spent picking the right fonts and font sizes would be for naught if you couldn't guarantee the font existed on the end user's PC. That's where font embedding comes in. It's easy to take an appropriately licensed font file and package it into your application for distribution to the target machines.

This chapter was about text, but we did dive into some things you might have considered to be controls (rich text, for example). In the next chapter you'll learn about the control model and how to use it in your apps along with binding and the MVVM pattern.

9

Controls, binding, and MVVM

This chapter covers

- Data and element binding
- The Model-View-ViewModel pattern
- Working with controls and lists
- Creating and using value converters

In previous chapters, you've learned how to display information on the screen, how to position it, and how to make it pretty. Those are all important skills to master to be an effective XAML developer or designer. What comes next is adding interaction both with the user and with data.

The first approach that most developers use when confronted with a new UI language or markup is to assign property values from code using simple `control.property = value` assignments. This is certainly familiar territory, but it doesn't leverage the power of the platform. It also creates too tight a coupling between the UI and the code behind it. The better approach is to use binding, but that has its own learning curve.

When learning binding, you can get into a lot of trouble by going about it in an unstructured way. It's relatively easy to bind to a property exposed in the code-behind, for example, but later you start running into issues where the exact binding path being used is next to impossible to figure out and debug. The code gets into a twisted knot, and you find yourself having to do things like manually setting contexts and bindings at different levels of the UI.

Most of the stumbling blocks in learning binding come from not having an appropriate backing structure. That's why MVVM was created. MVVM, or the Model-View-ViewModel pattern, is the de facto standard for structuring the XAML apps. It helps you set up a clean structure with separation between the UI and the data model and between the UI and supporting functionality. It improves testing and mocking (the ability to create dummy functionality or data), provides structure, and supports advanced patterns such as Inversion of Control (IoC). In addition to all that, it makes binding much easier to learn and much more pleasant to use.

Rather than teach you every possible property of every control and every nuance of the binding system, I'm going to show you how to set up something sustainable and usable for real apps. Throughout this chapter, we'll build out a contrived, albeit interesting, little app that simulates the remote control panel for a Mars rover. Figure 9.1 shows the app, complete with the controls we'll cover in this chapter.

We'll use MVVM to structure the app, of course. I'll show you how to work with several controls using binding and the MVVM pattern. We'll populate a `ListView` with data from an `ObservableCollection` and use data templates to format the display.

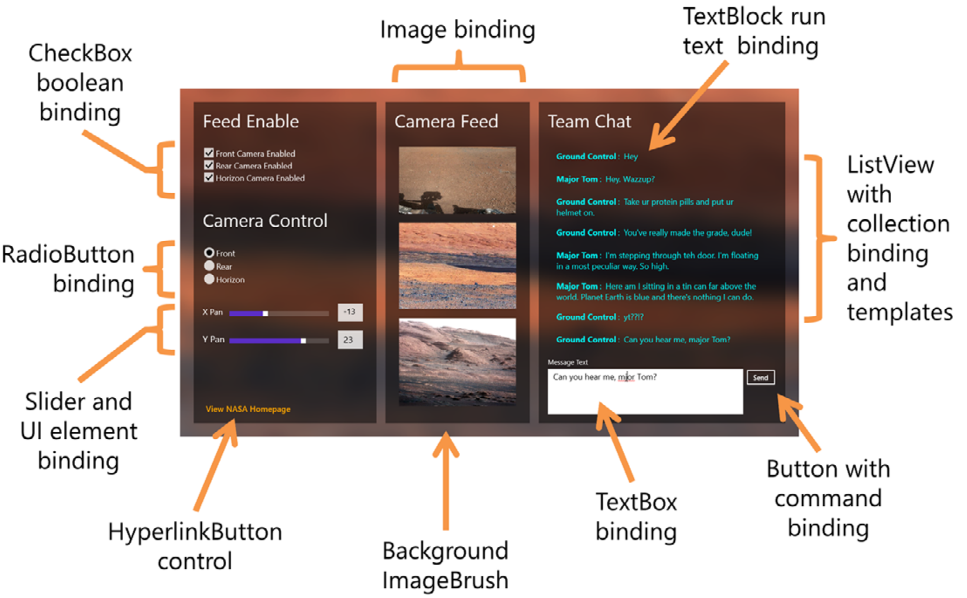


Image binding
TextBlock run text binding
CheckBox boolean binding
RadioButton binding
Slider and UI element binding
ListView with collection binding and templates
HyperlinkButton control
Background ImageBrush
TextBox binding
Button with command binding
Feed Enable
Camera Control
X Pan
Y Pan
View NASA Homepage
Camera Feed
Team Chat
Message Text
Send

☑ Front Camera Enabled
☑ Rear Camera Enabled
☑ Horizon Camera Enabled

● Front
○ Rear
○ Horizon

X Pan -13
Y Pan 23

Ground Control : Hey

Major Tom : Hey. Wazzup?

Ground Control : Take ur protein pills and put ur helmet on.

Ground Control : You've really made the grade, dude!

Major Tom : I'm stepping through teh door. I'm floating in a most peculiar way. So high.

Major Tom : Here am I sitting in a tin can far above the world. Planet Earth is blue and there's nothing I can do.

Ground Control : yt??!?

Ground Control : Can you hear me, major Tom?

Can you hear me, mjor Tom?

Figure 9.1 The app is a simulator of a rover remote control panel used by a large fictional space agency. It uses several different types of binding as well as several common UI controls and elements. It makes use of the brush and text information from the previous chapters, all tied together with binding and a healthy dose of MVVM.

We'll use two-way binding from text boxes, radio buttons, and check boxes and command binding from buttons. We'll show or hide images based on binding the `Visibility` property using a value converter to convert from a Boolean value. Along the way, I'll even show a little UI element-to-element binding.

9.1 The Model-View-ViewModel pattern

As with other platforms, there are many ways to accomplish the same things in XAML, most of which will get you into trouble two-thirds of the way through development. XAML apps scream out for some decent pattern for working with binding and UI controls.

The .NET + XAML world has generally adopted the MVVM pattern for separating the UI from the logic. MVVM was invented originally for WPF, as a variation of the Presentation Model pattern, so it has been road tested for the better part of a decade.

Figure 9.2 shows the essential parts of MVVM.

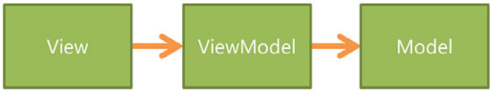

View
ViewModel
Model

Figure 9.2 The essence of MVVM, or as shown here, VVMM; so pure it's the Everclear (the drink/fuel/drain cleaner, not the band) of patterns.

If you're following what's shown in figure 9.2, you're "doing MVVM." That diagram doesn't provide the whole picture, though. Does the view really only talk to the viewmodel, or can it also talk with the model? Where does the data come from? What exactly is the communication between the layers? So, let's further break this out in a more typical implementation of the pattern. Figure 9.3 shows what that looks like, including the layers that can typically be tested using code-testing tools and approaches.

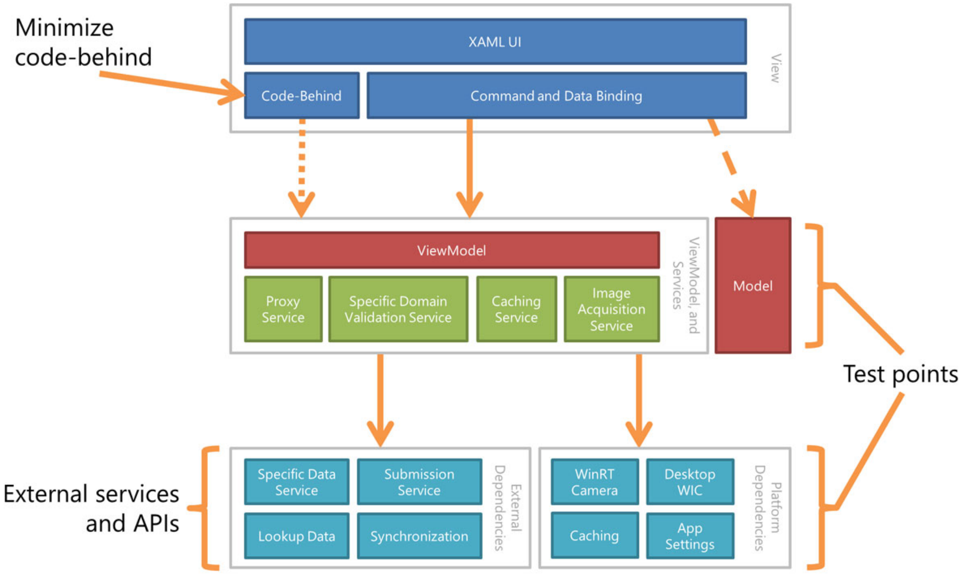

Minimize code-behind
XAML UI
Code-Behind
Command and Data Binding
View
ViewModel
Proxy Service
Specific Domain Validation Service
Caching Service
Image Acquisition Service
ViewModel, and Services
Model
Test points
External services and APIs
Specific Data Service
Submission Service
Lookup Data
Synchronization
External Dependencies
WinRT Camera
Desktop WIC
Caching
App Settings
Platform Dependencies

Figure 9.3 The MVVM pattern and its component parts. Don't get hung up on the details; just understand where the separation is and why it exists.

The *view* is the UI and the code that makes it hum. Any logic in the view is view-specific, handling animations, for example. Ideally, the code-behind is lean and mean, because it's extremely difficult to effectively test code that's stuck in the view. The dotted line between the code-behind and the viewmodel will be covered soon, but it represents the ability to interact with the viewmodel from code, event handlers, and more.

The next layer includes the *viewmodel* and a number of *services*. Services in this case are not web services or other remote bits but are instead internal proxies to other functionality. They're service classes. Each service class may abstract calls to web services, or they may call platform APIs or do more.

Some of those services will use the *external functionality* shown in the third layer. Although this isn't technically a core part of MVVM, it always comes into play in any nontrivial app. It's also an important layer to include if you intend to multitarget—Windows 8 and Windows Phone 8, for example, or Windows 8 and Silverlight or WPF.

Finally, you have the *model*. The model is generally made up of the entities in your app. There's a lot of flexibility here. In many implementations, the database interface is considered part of the model. In my own implementations over the years, I've always made the decision that model objects are simple containers for data, not containers for functionality or interfaces into other parts of the system. This has helped me create clean design each time.

There's a dotted line between the view and the model for good reason. There are two main schools of thought on how a model should be surfaced to the view:

- The underlying model should never be seen by the UI. Instead, you should create separate viewmodel entities that provide view-specific versions of the model entities.
- The viewmodel surfaces model entities directly.

There are good arguments for both. In the first approach, you don't have to worry about carrying over any gunk injected by an ORM, and you don't need to worry about polluting ORM-visible objects with binding-specific functionality. There are other ways around that, however, like the code-first Entity Framework. Because of that and because of the additional effort that would be involved in implementing option 1 in any nontrivial system, most implementations (including mine) use the second approach, or they standardize on the second approach except in places where the app developer sees a clear benefit to the first approach. You can mix and match.

I'll use the second approach throughout this book, where I surface model entities directly. Just understand that's not the only way, nor is it the best way in all situations. Do what works for your app, not what is in someone else's dogma.

Regardless of how far you intend to go with MVVM (events versus commands, different methods for locating viewmodels, and so on), there's a need for a pattern like this to focus our discussion of binding. Rather than spend half a chapter telling you the wrong way to do it, only to switch gears at the end, I figured we'd just create this up front. That does mean a little more of a learning curve here, but it's worth it, because it really is one of the best ways to structure your WinRT XAML apps.

Throughout the remainder of this section, we'll create the component parts to the solution to be used in this chapter. We'll start with the simplest, the model. This will contain classes that represent the data. Most binding statements will work with properties defined here.

The second piece will be the data service, which creates instances of the model classes. Technically part of the model, this will serve as a bit of a simplified data repository.

Next, we'll create the viewmodel. This will be the glue that, ahem, binds everything together. I consider the viewmodel to be the most important piece of the pattern and the one you should spend the most time internalizing. Luckily, viewmodels are simple beasts, similar to the classic Façade pattern, but with an XAML-friendly twist.

The final piece we'll create is the view. This is the UI to be used. It won't be wired up until later in this chapter, but we'll need some controls to use to interact with the viewmodel and model.

But before we do any of that, I want to introduce you to the MVVM toolkit we'll use throughout this chapter: MVVM Light.

9.1.1 Using an MVVM toolkit like MVVM Light

Most MVVM examples I've given at talks and in other books, I've done from scratch. But there are some things, like commands, that are just inconvenient to write from scratch. For those, I highly recommend using an MVVM toolkit. Each kit has its own approach to solving MVVM, and they each do things a little differently. Some provide a lot of infrastructure, some use conventions to wire up, and others simply cover the basics well.

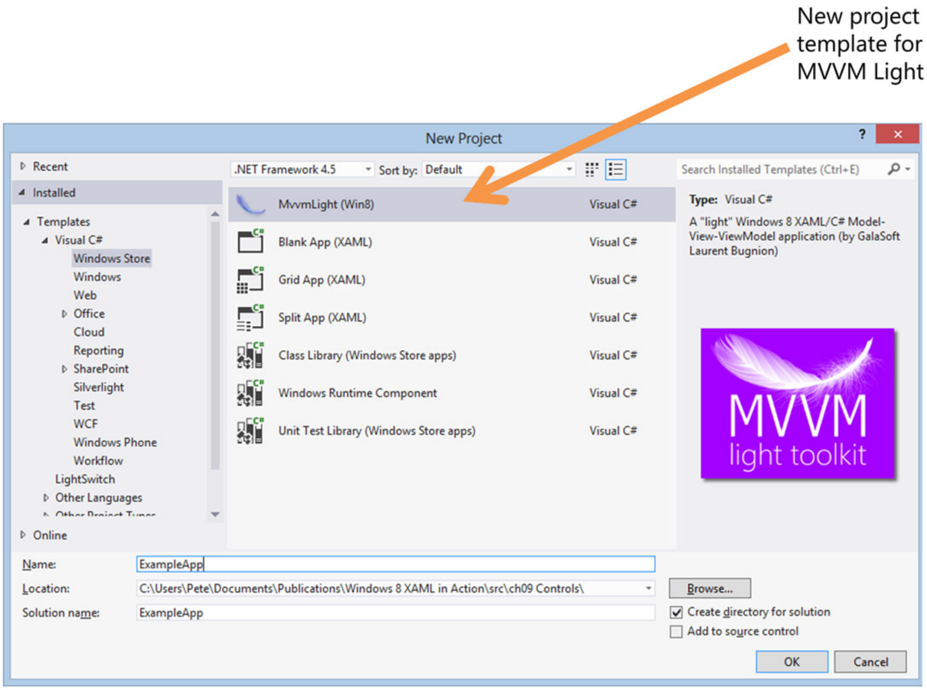
I'm not endorsing any particular kit, but because I had to pick only one, I chose MVVM Light by Laurent Bugnion. Laurent has been a fixture in the XAML community going back to WPF and Silverlight and has had his toolkit around and supported for all these platforms. I'm only going to use features that make it easier to learn the underlying technology, however, rather than those that may obscure what's really going on. MVVM Light is available on CodePlex at http://mvvmlight.codeplex.com/ and also via NuGet Package Manager from within Visual Studio.

INSTALLING THE TOOLKIT

For this chapter, I'm using MVVM Light Toolkit release 4.0.23.4, installed via the MSI on the MVVM Light download page on CodePlex. The toolkit is constantly updated, so your version numbers may be slightly different when you run through the examples here.

Run the MSI from the CodePlex page. Once the MSI has completed, you'll need to manually run the .vsix (Visual Studio Extension Installer) package as explained in the directions, which are displayed during installation. The readme is an HTML page with direct links to the .vsix packages, so this is easy to do.

Once the installation is complete, you'll have a new project type available to you, as shown in figure 9.4.


New project template for MVVM Light

New Project
? ×
Recent
Installed
Templates
Visual C#
Windows Store
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Windows Phone
Workflow
LightSwitch
Other Languages
Other Project Types
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
MvvmLight (Win8) Visual C#
Blank App (XAML) Visual C#
Grid App (XAML) Visual C#
Split App (XAML) Visual C#
Class Library (Windows Store apps) Visual C#
Windows Runtime Component Visual C#
Unit Test Library (Windows Store apps) Visual C#
Type: Visual C#
A "light" Windows 8 XAML/C# Model-View-ViewModel application (by GalaSoft Laurent Bugnion)
MVVM
light toolkit
Browse...
OK
Cancel

Name: ExampleApp
Location: C:\Users\Pete\Documents\Publications\Windows 8 XAML in Action\src\ch09 Controls\
Solution name: ExampleApp
☑ Create directory for solution
☐ Add to source control

Figure 9.4 The New Project dialog with the MvvmLight template shown

CREATING THE PROJECT

Create a project using this template. Name it `ExampleApp` to stay consistent with the examples in this chapter. Once the project has been set up, run it to see what it includes.

There are a number of pieces that are included in the MVVM project. As mentioned earlier, I won't use all of them in this chapter, because it's more important that we focus on learning controls and binding, leaning on the toolkit only when it helps us get to that goal more quickly. I won't use interfaces and viewmodel locators in my examples. Once you understand the base pattern, your understanding of those (and why you may want those) will be highly dependent on which toolkit you pick.

9.1.2 *The model*

The model is a representation of the data in the app. Collectively, the data entities and the way they're created or populated is all considered part of the model. Technically, some of those pieces may live in services or out of the app, but conceptually they're part of the model.

I'm a big fan of dumb model objects, that is, model objects that are simply entities or containers for data, without any real functionality. I don't care for big object-oriented model trees, but some folks do like them. I want the model as simple as possible: something for me to pass around, bind to, and maybe send back to the database.

In the Model folder, create a class named `ChatMessage`, and in that class, add the following code.

Listing 9.1 The `ChatMessage` model class in its initial form

```
namespace ExampleApp.Model
{
  public class ChatMessage
  {
    public int Id { get; set; }
    public string From  { get; set; }
    public string Message  { get; set; }
  }
}
```

Properties to bind to

This is the first version of the `ChatMessage` class. You'll make a few changes to this later in the chapter. Now you just need a way to create a bunch of model instances for use in the app. For that, you'll use a data service.

9.1.3 The chat data service

The only service you'll use in this chapter is a data service. In your apps, this might go out to a repository and pull in data, maybe call a web service or three, or pull from a local SQLite database.[1] The focus of this chapter isn't on that, so the data is simply dummy data in the class.

MVVM Light includes great support for IoC and for pulling in different data classes. As I mentioned earlier in this chapter, that type of functionality gets in the way of what I want to show, so I'm going with a straight data class, no interfaces, and, when it comes time to instantiate it, hardcoded references. Just in case you were still planning on sitting quietly at home instead of marching on my home with torches and pitchforks, I've also made the class a singleton. A *singleton* is a class that ensures that only one instance (sometimes one per thread depending on the implementation) of itself is alive at any point in time. For more on the singleton pattern, please see this page: http://bit.ly/WikiSingleton.

Inversion of Control

IoC is a way to couple and substitute classes at runtime rather than at compile time. For example, you may want to have different data classes for your test app and your production app, but you don't want to change the code when switching between them. Similarly, you might do the same to enable your designer to work on the UI without having the entire infrastructure behind them.

IoC typically uses Dependency Injection (DI) to pull in the class instances at runtime. Through this, the classes can be developed independently, without tight dependencies between each other.

---

[1] Yes, you can use local databases from WinRT XAML apps. Awesome! See http://www.sqlite.org/.

(continued)

As MVVM started to mature, many developers linked IoC with MVVM in such a way as to imply that if you weren't using IoC, you "weren't doing MVVM right."

IoC and DI aren't requirements for working with the MVVM pattern. They do complement it nicely, though. Because of this, many MVVM toolkits have built-in IoC containers right in the codebase or in a state-recommended third-party container in the documentation.

In small apps, these are often overkill. Used improperly or inconsistently, you can make a complete mess of an app with IoC. But in larger apps, you can quickly see the benefits of using these features. Just make sure you fully understand the principles and uses of IoC and DI and don't implement them simply because someone else told you to.

The following listing has the `ChatDataService` class, created in the Model folder of the project.

Listing 9.2 The `ChatDataService` class

```
using System.Collections.Generic;

namespace ExampleApp.Model
{
  public class ChatDataService
  {
    private static ChatDataService _current = null;
    public static ChatDataService Current
    {
      get
      {
        if (_current == null)
          _current = new ChatDataService();

        return _current;
      }
    }

    public IList<ChatMessage> GetMessages()
    {
      var messages = new List<ChatMessage>();

      messages.Add(new ChatMessage()
      { Id = 1, From = "Ground Control",
        Message = "Hey"});

      messages.Add(new ChatMessage()
      { Id = 2, From = "Major Tom",
        Message = "Hey. Wazzup?" });

      messages.Add(new ChatMessage()
      { Id = 3, From = "Ground Control",
        Message = "Take ur protein pills and put ur helmet on." });
```

Singleton support

Temporary list

Construct one ChatMessage

```
        messages.Add(new ChatMessage()
        { Id = 4, From = "Ground Control",
          Message = "You've really made the grade, dude!" });

        messages.Add(new ChatMessage()
        { Id = 5, From = "Major Tom",
          Message = "I'm stepping through teh door. " +
            "I'm floating in a most peculiar way. So high." });

        messages.Add(new ChatMessage()
        { Id = 6, From = "Major Tom",
          Message = "Here am I sitting in a tin can far above the world. " +
              "Planet Earth is blue and there's nothing I can do."} );

        messages.Add(new ChatMessage()
        { Id = 7, From = "Ground Control",
          Message = "yt??!?" });

        return messages;                              Return list
      }
    }
}
```

I considered inflicting "Rocket Man" on you, knowing full well that in your head, you'd hear it in the pretentious William Shatner rendition, but I wanted to minimize book returns. Instead, you have here a chat version of Bowie's 1969 classic "Space Oddity." I'll just wait here while you go dig it up on YouTube or iTunes.

The `ChatDataService` uses a singleton pattern and includes a single function that returns the dummy data. It doesn't include any functions for adding or deleting data, although a real data service almost certainly would. This data service is called from the viewmodel to populate the viewmodel's copy of the data. No transformation is applied to the data, but the viewmodel is where you could sort, filter, or otherwise make the data fit the view's requirements.

The `ChatDataService` class will be called from only one place in your app: the viewmodel.

9.1.4 The MainViewModel and CameraViewModel classes

This is where it all hangs together. This is where you call the data service; this is where you'll provide properties and collections for the UI to bind to. The UI will never speak directly to the data service or to any other services. Instead, the viewmodel will offer an interface to that, abstracting it away. It will also provide a place to safely call functions from the UI or, as you'll see later, execute commands.

In the ViewModel folder, create a new class named `CameraViewModel`. This will represent the output from one of the fictional rover cameras, so it includes an output image, as well as a few configuration properties. The next listing has the code for this version of the class.

Listing 9.3 The first iteration of the `CameraViewModel` class

```
using GalaSoft.MvvmLight;
using Windows.UI.Xaml.Media;
```

```
namespace ExampleApp.ViewModel
{
  public class CameraViewModel : ViewModelBase
  {
    public string Name { get; set; }
    public int XPosition { get; set; }
    public int YPosition  { get; set; }
    public bool IsEnabled  { get; set; }
    public ImageSource Image  { get; set; }
  }
}
```

Bindable properties

The `XPosition` and `YPosition` properties will be used for simulating panning of the camera. The `IsEnabled` property decides whether the camera's output will be displayed, and the `Image` property has the actual image to display.

The class inherits from `ViewModelBase`, provided by the MVVM Light Toolkit. As you'll see later, that provides functionality around change tracking. It also provides a number of great features you won't need in this app, such as design mode support, messaging, and more.

But wait, why is this class a viewmodel class and not simply a model class? In reality, you could make a case either way. But if you consider that this class may in the future include functionality for actually panning something or otherwise performing actions, it doesn't fit my definition of a model class. It's also something that isn't really data; it's an abstraction that allows you to have several instances of this available to the form without cluttering up the main viewmodel. It does, or may, represent more than the state of the camera, so it's more than a model object.

The following listing has the first iteration of the `MainViewModel` class. There's already one in the solution; replace its contents with this simplified version.

Listing 9.4 The initial form of the `MainViewModel` class

```
using ExampleApp.Model;
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using Windows.UI.Xaml.Media.Imaging;

namespace ExampleApp.ViewModel
{
  public class MainViewModel : ViewModelBase
  {
    public MainViewModel()
    {
      _cameras = new CameraViewModel[3];
      for (int i = 0; i < _cameras.Length; i++)
      {
        _cameras[i] = new CameraViewModel();
      }
```

ViewModelBase

Initialize cameras and images

```
      _cameras[0].Name = "Front camera";
      _cameras[1].Name = "Rear camera";
      _cameras[2].Name = "Horizon camera";

      LoadImages();
    }

    private CameraViewModel[] _cameras;
    public CameraViewModel[] Cameras
    {
      get { return _cameras; }
    }

    private void LoadImages()
    {
      _cameras[0].Image = new BitmapImage(
         new Uri("ms-appx:/Assets/simulated_front_cam.jpg"));
      _cameras[1].Image = new BitmapImage(
         new Uri("ms-appx:/Assets/simulated_rear_cam.jpg"));
      _cameras[2].Image = new BitmapImage(
         new Uri("ms-appx:/Assets/simulated_horizon_cam.jpg"));
    }

    public void LoadMessages()
    {
    }
  }
}
```

Initialize cameras and images

CameraViewModel instances

LoadMessages to be provided later

We've previously covered the `Image` element, so the support for that is going to be baked into the viewmodel (and the view) from the start. Support for the rest of the controls will come in the remaining sections of this chapter.

Contrary to many first-time implementations I've seen, the viewmodel should not be considered just a replacement for the code-behind. Don't make a big, fat viewmodel full of all the code you yanked from a .xaml.cs file. Instead, treat the viewmodel more like a light, page-specific façade into the model and the various service classes. Remember, code stuck in a viewmodel can't be reused across pages in a one-viewmodel-one-page typical MVVM architecture, and fat viewmodels are difficult to test and change.

When designing apps, I often start with the view, because I'm a UI guy. That doesn't work super well when writing a book with code for you to type in, however, so I've left the view as the last piece to add.

9.1.5 The view

The view is the UI; it's where all the controls will live and where you put into practice the layout, text, and other knowledge you've gained so far. The view includes not only the XAML I'll show in this section but any code-behind as well.

It's difficult to automate testing of the view. There are tools to do it, and people you can recruit to bang on keys, but the effort involved is often much higher than

API-level testing and not usually incorporated into any source control check-in gates. For those reasons, most developers recommend that your view's code-behind contain as little code as possible. In reality, what they're saying is, "The code-behind code isn't really going to be tested, so don't stick anything important there."

Some will tell you that the code-behind must have absolutely no code, but I don't believe in absolutes and try not to be religious. Put code there, but keep it minimal and trivial. Make it UI-specific code (for working with animations, for example, or setting the data context), but don't fill your code-behind with logic and other stuff that you really want to test. Put that in the viewmodel and service classes instead.

That all said, we'll get to the code-behind later, when we have something to actually put in it. For now, create the start of the UI using the XAML in the listing that follows. You'll need the downloadable images for this chapter (or some substitutes) for the background and, later, for the camera images.

Listing 9.5 The skeleton view XAML for MainPage.xaml

```
<Page x:Class="ExampleApp.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:ignore="http://www.ignore.com"
  mc:Ignorable="d ignore"
  d:DesignHeight="768"
  d:DesignWidth="1366">

  <Grid>
    <Grid.Background>
      <ImageBrush ImageSource="ms-appx:/Assets/rover_controller_bg.png" />
    </Grid.Background>

    <Grid Margin="20 20 20 20">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width=".75*" />
        <ColumnDefinition Width=".6*" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Grid Grid.Column="0" Margin="10">
        <Rectangle Fill="#000000" Opacity="0.5"/>
        <StackPanel Margin="20">
          <TextBlock Text="Feed Enable" FontSize="40"
                     Margin="0 0 0 30"/>

            <!-- Camera Enable Checkboxes Go Here -->

          <StackPanel Margin="0 60 0 30">
            <TextBlock Text="Camera Control" FontSize="40"
                       Margin="0 0 0 30"/>

              <!-- Radio Buttons Go Here -->
          </StackPanel>
```

CheckBoxes

RadioButtons

```
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto" />
          <ColumnDefinition Width="*" />
          <ColumnDefinition Width="75" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <TextBlock Text="X Pan"
                   VerticalAlignment="Center"
                   FontSize="18"
                   Grid.Column="0"
                   Grid.Row="0" />

          <!-- Slider UI goes here -->

      </Grid>
    </StackPanel>

  <HyperlinkButton x:Name="ViewOnlinePage"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Bottom"
                   Foreground="Orange"
                   Margin="10"
                   FontSize="18"
                   NavigateUri="http://www.nasa.gov"
                   Content="View NASA Homepage" />
</Grid>

<Grid Grid.Column="1" Margin="10">
  <Rectangle Fill="#000000" Opacity="0.5"/>
  <StackPanel Margin="20">
    <TextBlock Text="Camera Feed"
               Margin="0 0 0 20"
               FontSize="40"/>

          <!-- Images go here -->
  </StackPanel>
</Grid>

<Grid Grid.Column="2" Margin="10">
  <Rectangle Fill="#000000" Opacity="0.5"/>
    <Grid Grid.Margin="20">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>
```

Sliders

Images

```
          <TextBlock Text="Team Chat"
                     Margin="0 0 0 20"
                     Grid.Row="0"
                     Grid.Column="0"
                     Grid.ColumnSpan="2"
                     FontSize="40"/>

              <!-- Message List Goes Here -->

          <TextBlock Text="Message Text"
                     Margin="0 10 0 5"
                     FontSize="15"
                     Grid.Column="0" Grid.Row="2"/>

              <!-- Message TextBox Goes Here -->

          <Button x:Name="SendMessage"
                  VerticalAlignment="Top"
                  Margin="5 0 0 0"
                  Content="Send"
                  Grid.Column="1"
                  Grid.Row="3" />
        </Grid>
      </Grid>
    </Grid>
  </Grid>
</Page>
```

ListView

TextBox

Note the various placeholders in the XAML. Throughout the remainder of the chapter, you'll replace these with chunks of XAML that use specific controls to build up the UI.

The view is just a bit of a skeleton at the moment. Not only are the controls not there, but nothing connects the view to the viewmodel. Why is that? You need to understand binding before that can happen, because binding is where all the magic will happen.

9.2 Binding primer

It seems that every language and platform have their own way of getting values into properties of UI components. Most of them support direct `lvalue = rvalue` type assignment, and most also now support binding. XAML was built from the ground up to make binding a core part of the infrastructure so that you almost never have to do something like `SomeTextBox.Text = SomeObject.Property`. After the vector graphics, binding was the thing that really excited me when I first learned XAML. I was coming from a world where *binding* was a dirty word, and WPF and Silverlight were working to turn that around.

But binding is more than just getting data into UI components: It's also about getting data back out of them. Even more than that, it lets you do the same between UI components so you can bind a property of one to a property of another.

In a nutshell, binding is the mechanism for updating one property from another. Most of the time you'll have a UI element like a `TextBlock` or `TextBox` bound to some sort of data object, like a customer or a message, so that will be the major focus of this section.

Throughout this section we'll look at what makes binding work and what you need to understand to be able to use binding in the application. First, we'll look at the binding source and target, to understand the relationship between these two objects. Error messages and documentation will often refer to source and target explicitly, so it's important to know the distinction.

Then, we'll look at how to control the direction and frequency of the flow of updates between the source and target. This is easily done using the `Mode` property of the `Binding` object.

Data that doesn't change is not all that interesting. UI controls that don't update when the data updates are not just boring; they're buggy. For those reasons, we'll next look at how to handle change notification, including some of the goodies built into the MVVM Light Toolkit.

Finally, when looking at a binding statement, you need to know what the paths are relative to. You also need a way to set this globally for a page, or container, or even a small group of elements. For that, we'll look at the `DataContext`.

Let's start with understanding the objects in play and get a little terminology straight.

9.2.1 The source and target

Each binding relationship is made up of two sides: the source and the target. That's further broken down into object and property on each side. The target object is typically a UI element, such as a `TextBlock` or a `TextBox`. Except in the case of UI element binding (or binding in code), the target is always the object with the binding statement in markup. More correctly, the target is the first property that will be set, whereas the source is the first property that will be read. What happens after the first time depends on the binding mode.

The target property must be a dependency property; this is because binding relies on the ability of a dependency property to source its value externally, as described in chapter 4. Figure 9.5 shows this relationship using the `Binding` markup extension.

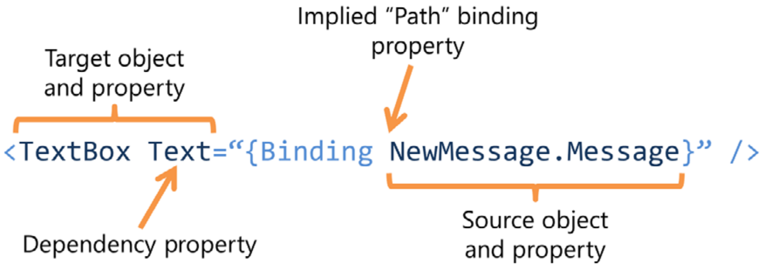

Implied "Path" binding property
Target object and property
<TextBox Text="{Binding NewMessage.Message}" />
Dependency property
Source object and property

Figure 9.5 The relationship between the binding source and the binding target. The target object is usually a UI element, and its target property must be a dependency property. The source object can be just about anything, as can the source property. Most people leave out the `Path=` part of the statement, because that can be easily inferred.

The source object may be just about anything, but typically, it's going to be a data object, a viewmodel, or sometimes even another control. In this chapter, the `ChatMessage` class instance and `MainViewModel` instance are two examples of binding sources.

The source property may also be anything that's compatible with the target property. What defines compatibility? It's decided by straight type casting, built-in .NET type converters, and, if provided, value converters. We'll look at an example value converter later in this chapter.

> **TIP** The binding source property must always be a property. A field, even a public field, cannot be a binding source. That's why even the initial version of the `ChatMessage` class used properties. Your binding will fail silently if you use a plain-old field like `public int foo;`. Autoproperties (properties followed by `{ get; set; })` work for binding but don't have change notification. For that, you'll need to break them out into complete properties, backed by field.

Importantly, the source object must provide some way of notifying when the source property changes, if you want to update at all beyond the initial read, or a `OneTime` binding mode.

9.2.2 Binding mode

As a developer, you may need to control the direction in which data flows between the source and the target. In most cases, this is simply to enable functionality, but in others it may be for performance reasons or to use a control in a nonstandard way. In the binding statement, this behavior is specified by using the `Mode` property, as shown here:

```
<TextBox Text="{Binding NewMessage.Message, Mode=TwoWay}" />
```

There are three possible values for the `Mode` property:

- `OneTime`—The binding target is updated only once when the binding is created.
- `OneWay`—The binding target is updated each time the source property changes, assuming the source is properly notifying of the property change.
- `TwoWay`—The binding target is updated each time the source property changes, again assuming change notification is in place. Additionally, the binding source is updated each time the target property changes. This is most commonly used to get user-entered text back into binding sources, but it may also be used for sliders and other types of controls.

It's extremely rare to find anyone using the `OneTime` value, but in a performance-sensitive system, this can cut down the load on the binding infrastructure. By far, the most common approach is `OneWay`, which is the default in cases where no `Mode` is specified.

As I mentioned, updating beyond the initial read requires change notification; let's look at that.

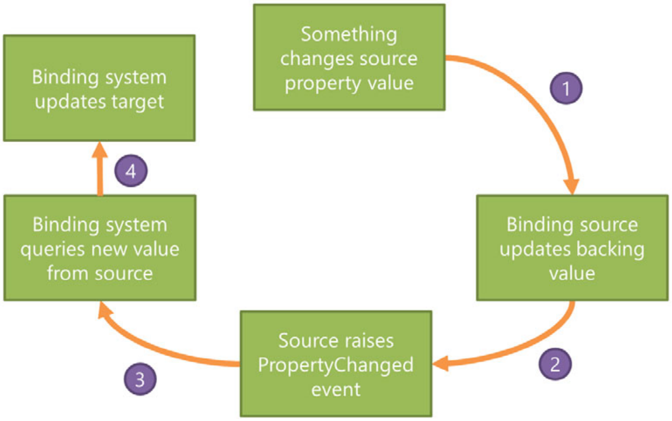

Something changes source property value
1
Binding source updates backing value
2
Source raises PropertyChanged event
3
Binding system queries new value from source
4
Binding system updates target

Figure 9.6 When the binding source implements the `INotifyPropertyChanged` interface, this is how change notification and target updates work.

9.2.3 Change notification

If you want the target property to always reflect the current value of the binding source property, you need to somehow notify the binding system when the source property changes. If the source property is a dependency property, when binding to other controls, for example, this is handled automatically. In cases where the binding source is a data object of some sort—a viewmodel or model object as it is in this case—you need to manually notify the binding system of any value changes. This is accomplished through the `INotifyPropertyChanged` interface or, in the case of an MVVM Toolkit like Laurent's, via the `ObservableObject` base class. In either case, what's happening is each time a property changes in the source, you raise an event with the name of the property as part of one of the arguments. The binding system catches this event and knows to requery the value when it receives it. Figure 9.6 shows the relationship between the objects and the events.

Listing 9.6 shows the updated `ChatMessage` class, which now implements `INotify-PropertyChanged` (INPC) through the `ObservableObject` base class. I'm only going to show a single property, because there's another inline change notification approach we'll use in the final version. By the way, notifying properties like this is a good excuse to dust off your code-snippet-creation skills, because the properties all follow a well-defined pattern.

Listing 9.6 The `ChatMessage` class with change notification

```
using GalaSoft.MvvmLight;

namespace ExampleApp.Model
{
  public class ChatMessage : ObservableObject
  {
    private int _id;
    public int Id
    {
```

MVVM Toolkit namespace

ObservableObject implements INPC

```
   get { return _id; }
   set
   {
    _id = value;
    RaisePropertyChanged("Id");
   }
  }

...
 }
}
```

Raise PropertyChanged event

Gained through the inheritance from `ObservableObject`, the `RaisePropertyChanged` method call notifies any listeners, including the binding system and anything explicitly subscribed to the event, that the property value has changed. Underneath, this is accomplished via the `INotifyPropertyChanged.PropertyChanged` event and its `PropertyChangedEventArgs` class, but the base class thinly insulates you from that. You call this method from the setter so you can guarantee that it fires every time a value is assigned to the property.

Note that, in the end, property change notifications always specify the property name as a string; that's how the binding system works. But there are several interesting ways to avoid magic strings in your code, using lambda expressions. The MVVM Light Toolkit has built-in support for using lambda expressions to get compile-time checking in the change notification, something you can't get when using strings.

If your system's usage pattern is such that properties may be set again and again with the same value, you should check to see if the value actually changed before setting the field and raising the event.

Let's do that and also clear out the use of strings from the code. The next listing shows how to use another MVVM Light feature to handle everything in a single function call.

Listing 9.7 Using built-in support for value checking and change notification

```
using GalaSoft.MvvmLight;

namespace ExampleApp.Model
{
 public class ChatMessage : ObservableObject
 {
  private int _id;
  public int Id
  {
   get { return _id; }
   set { Set<int>(() => Id, ref _id, value); }
  }

  private string _from;
  public string From
  {
   get { return _from; }
```

ObservableObject

Backing fields

Set method call (the <type> is optional)

```
      set { Set<string>(() => From, ref _from, value); }
    }

    private string _message;
    public string Message
    {
      get { return _message; }
      set { Set<string>(() => Message, ref _message, value); }
    }
  }
}
```

Backing fields

Set method call (the <type> is optional)

The MVVM Light Toolkit's `ObservableObject` base class includes a few versions of a method named `Set`. This method takes in a lambda expression that points to the property itself, a reference to the backing property (so it has something to set), and the new value. By removing the use of strings, you now get compile-time verification that you have the correct property names in use. Few things are worse than trying to troubleshoot a binding problem only to find out the root cause was a typo in the string passed to a `RaisePropertyChanged` call.

Inside the `Set` method, it does the appropriate checking to see if the property value is different from the old value. If so, it uses the lambda expression to figure out the property name and then raises the property changed notification after setting the backing field value.

The lambda expression approach isn't quite as fast as using strings, because it must reflect on the model to convert the name to a string, but it sure makes for much more airtight code.

Let's apply the same approach to the `CameraViewModel` class. The updated version of this class is shown here.

Listing 9.8 Updated version of the `CameraViewModel` class

```
using GalaSoft.MvvmLight;
using Windows.UI.Xaml.Media;

namespace ExampleApp.ViewModel
{
  public class CameraViewModel : ViewModelBase
  {
    private string _name;
    public string Name
    {
      get { return _name; }
      set { Set<string>(() => Name, ref _name, value); }
    }

    private int _xPosition = 0;
    public int XPosition
    {
      get { return _xPosition; }
      set { Set<int>(() => XPosition, ref _xPosition, value); }
    }
```

Set and notify

```
    private int _yPosition = 0;
    public int YPosition
    {
      get { return _yPosition; }
      set { Set<int>(() => YPosition, ref _yPosition, value); }
    }

    private bool _isEnabled = true;
    public bool IsEnabled
    {
      get { return _isEnabled; }
      set { Set<bool>(() => IsEnabled, ref _isEnabled, value); }
    }

    private ImageSource _image;
    public ImageSource Image
    {
      get { return _image; }
      set { Set<ImageSource>(() => Image, ref _image, value); }
    }
  }
}
```

Set and notify

This listing does for the `CameraViewModel` what the previous listing did for the `ChatMessage`. It augments the class to ensure change notification happens properly. Without this, any two elements bound to the same property would show different values when one was changed, because the other would never be notified of the update.

When working with a viewmodel, especially the page-wide viewmodel, the markup typically doesn't reference the viewmodel itself. Instead, it references properties on the viewmodel. But how does the page know to start at the viewmodel when resolving those property names? That's where the `DataContext` comes into play.

9.2.4 DataContext

In a complex system, fully qualifying the path to the binding source can be cumbersome if not impossible. Consider a situation where you have nested viewmodels that expose model objects that, in turn, are composite objects. The path gets unwieldy. Now consider throwing a collection into the middle of that or having to deal with creating instances of objects, and it becomes a complete mess, if not impossible in some cases.

The concept of a data context helps save you from that. Every element in XAML has a `DataContext` property. This is a reference to a binding source object. Every binding statement made within the scope of that element (think of a `Grid`, for example) will be relative to the `DataContext`.

One really common way to use the `DataContext` is to set the entire page's `DataContext` to the associated viewmodel instance. By doing this, as shown in the next listing, every binding statement on the page will be relative to the viewmodel.

Listing 9.9 Setting the data context from the code-behind

```
using ExampleApp.ViewModel;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
```

```
namespace ExampleApp
{
  public sealed partial class MainPage : Page
  {
    private MainViewModel _vm = new MainViewModel();

    public MainPage()
    {
      this.InitializeComponent();

      DataContext = _vm;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
      _vm.LoadMessages();
    }
  }
}
```

Create viewmodel

Set data context

The listing shows how to set the `DataContext` from the code-behind. By default, the MVVM Light Toolkit will use a viewmodel locator to dynamically associate the viewmodel based on naming convention, but that obscures the importance of the data context. By all means, however, use it in a real app, because it's a great feature.

Note that in this listing you're also loading the message data. You do this from the `OnNavigatedTo` event, which is fired when the page is navigated to by the user. The viewmodel code handles dup checking in cases where this page is navigated to back and forth. A further optimization would be to set a flag in the viewmodel (or data service) and not perform any load if the data is already present. In a system where you're making a network or other high-latency call, this can be an important consideration.

Binding is one of the most useful features of XAML-based apps. It cuts down on the amount of code you need to write to get and set values, and it really helps decouple the UI from the underlying model. With the assistance of the MVVM pattern, you can create a UI that, with perhaps the exception of setting the data context from code-behind, if you go that route, has no in-code references to other layers of the app. This makes peeling off the UI for testing or porting purposes something that mere mortals are able to accomplish.

In the next section, you'll use this new-found binding knowledge to get some data to and from controls, starting with the most common: text.

9.3 Entering and displaying text

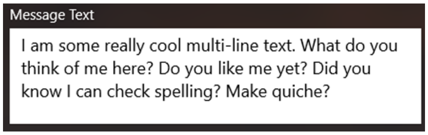
Most nontrivial apps require the user to type something at some point. Whether that's a comment on an entry, a message to someone, or some login information, you still need the user to get in front of a physical or virtual keyboard and bang away at some characters. Ever since I worked in Silverlight 1.1a, which didn't include a `TextBox` of any sort, I've made sure to never take built-in text entry for granted.

With all the binding and MVVM plumbing out of the way, it becomes quite easy to work with controls. One of the most common controls is the `TextBox`, which lets you enter single or multiline text using a single font style. A variation on the `TextBox`, the `PasswordBox` enables you to capture sensitive information like, you guessed it, a password while masking the contents. It follows the Windows 8 conventions including the "press to reveal" button to the right of the box. Every time you log into Windows using a text password, you're using a native password box.

With the exception of the `RichEditBox`, which isn't at all binding friendly without outside additions,[2] each of these text-editing controls works well in an MVVM-based solution. In the context of our solution, let's look at the first two: the `TextBox` and the `PasswordBox`.

9.3.1 Working with the TextBox

When you want the user to enter text, whether through a physical keyboard or an onscreen touch keyboard, the control you need to use is the `TextBox`. With the exception of multiline text wrapping and some productivity features, this control hasn't changed much from what we've been using in any other platform over the years. Figure 9.7 shows the `TextBox` used in our app.


Message Text

I am some really cool multi-line text. What do you think of me here? Do you like me yet? Did you know I can check spelling? Make quiche?

Figure 9.7 The humble `TextBox` with a `TextBlock` label above it

The entered text in the `TextBox` is stored in the `Text` property. This is a dependency property with a property wrapper, so you can use binding or straight code assignment to set and get the value. The next listing shows the markup segment with the `TextBox` for entering a chat message. Place this XAML in the spot you reserved for the chat text entry.

Listing 9.10 `TextBox` for chatting with other team members in faraway places

```
<TextBox Grid.Column="0" Grid.Row="3"
         Text="{Binding NewMessage.Message, Mode=TwoWay}"
         TextWrapping="Wrap"
         FontSize="18"
         Height="100"/>
```

TwoWay binding

Note the use of `TwoWay` binding here. That allows the `TextBox` to be updated from the viewmodel and the viewmodel to be updated from the `TextBox`. With `TextBox` controls, you almost always want to use `TwoWay` binding.

The binding source is an instance of the `ChatMessage` class exposed through a `NewMessage` property. This is the same class you're returning instances of from the data service. The following listing has the `NewMessage` property for the `MainViewModel` class.

---

[2] The community is still working on best approaches for working with the `RichEditBox`. I've seen implementations using attached properties to enable binding, for example. All of the versions are messy, and none of them get my endorsement just yet.

Listing 9.11 `ViewModel` support for the new message binding

```
private ChatMessage _newMessage;
public ChatMessage NewMessage
{
  get { return _newMessage; }
  set { Set<ChatMessage>(() => NewMessage, ref _newMessage, value); }
}

private void CreateNewMessage()
{
  if (NewMessage != null)
    NewMessage.PropertyChanged -= NewMessage_PropertyChanged;

  NewMessage = new ChatMessage();
  NewMessage.PropertyChanged += NewMessage_PropertyChanged;
}
void NewMessage_PropertyChanged(object sender, PropertyChangedEventArgs e)
{ }
```

ChatMessage

Unwire old event

Wire new event

We'll use this later

This looks similar to other code you've added. It's simply providing a place on the viewmodel that you can use as a binding source for the `TextBox`. Now, the `TextBox` can always be typed into, and you'll always have a place where that data will end up.

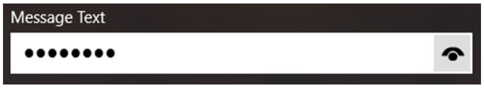
The `CreateNewMessage` function is included here, which performs some necessary cleanup you'll use later. It also creates the `ChatMessage` instance, so in the `MainViewModel` constructor, be sure to add the following bit of initialization code. Otherwise, the `TextBox` will be bound to a null instance, which isn't going to work out well for you.

```
public MainViewModel()
{
  ...
  CreateNewMessage();
}
```

Once you have that call in the constructor, you should be able to fire up the UI and type away in the `TextBox`. If you put a breakpoint into the `Text` setter in the `ChatMessage` class, type in the `TextBox` and tab away; you'll see the data make it into the class.

9.3.2 *Experimenting with the PasswordBox*

Now, for grins, let's change the `TextBox` to a `PasswordBox`. Save the project, first, because you won't stick with the results of this little side trip. The markup won't be the same, because the `PasswordBox` doesn't include a `Text` property. Instead, it has a `Password` property. It also has an `IsPasswordRevealButtonEnabled` property, which you can set to `true` to show the little reveal button on the right, as shown in figure 9.8. This is especially useful in tablets where your muscle memory isn't quite as reliable as on a physical keyboard.



••••••••

Figure 9.8 The `PasswordBox` with password reveal button visible

Although to remain consistent I don't recommend changing it, the control also includes a `PasswordChar` property, which allows you to change the character that's displayed in place of the typed characters. The following listing shows the changes required to support the `PasswordBox` in place of the `TextBox`.

Listing 9.12 Playing around with the `PasswordBox`

```
<PasswordBox Grid.Column="0" Grid.Row="3"
             IsPasswordRevealButtonEnabled="True"
             Password="{Binding NewMessage.Message, Mode=TwoWay}"
             FontSize="18"/>
```

This short listing includes the markup for the `PasswordBox`. Note that although the property was changed from `Text` in the `TextBox` to `Password` in the `PasswordBox`, the binding statement itself remains the same.

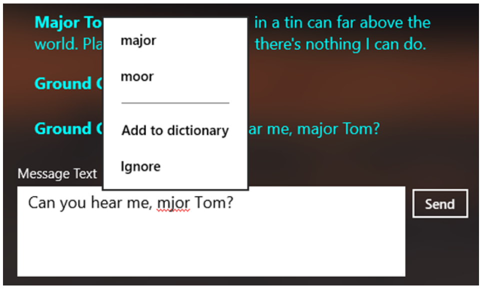
If you haven't already, go ahead and revert to the `TextBox` markup now, so we can look at productivity features of the control.

9.3.3 Spell checking and autocorrect

Every modern browser has built-in spell checking in its text-entry fields. It's become both necessary and expected. Users have been asking for this in their desktop apps for, quite literally, decades. In most cases, we've had to use third-party solutions or cobble together things that just weren't system native and always felt like one-off solutions.

One great thing about WinRT XAML is that the platform supports system-driven spell checking and both autocorrect and autocomplete. This is something WPF developers had (well, sort of) and Silverlight developers wanted but were unable to get. I'm really excited to see this feature baked into WinRT from day one.

Figure 9.9 shows the spelling context menu with a normal `TextBox`. Notice also the standard underlining of the misspelled word.



Major To… in a tin can far above the world. Pla… there's nothing I can do.

Ground C…

Ground C… …ar me, major Tom?

Can you hear me, mjor Tom?

Figure 9.9 Even the plain-old `TextBox` control supports spell checking and autocorrect.

To support spell checking and autocorrect, rather than using some one-off implementation for XAML, it uses system dictionaries and correction rules. All you have to do as a developer is turn it on using two simple properties, as shown here.

Listing 9.13 Enabling spell checking and autocorrection in the `TextBox`

```
<TextBox Grid.Column="0" Grid.Row="3"
         Text="{Binding NewMessage.Message, Mode=TwoWay}"
         IsSpellCheckEnabled="True"
         IsTextPredictionEnabled="True"
         TextWrapping="Wrap"
         FontSize="18"
         Height="100"/>
```

Enable spell checking

Enable autocorrection

As you can see in this listing, supporting these seemingly advanced features takes almost no work on your part. I encourage you to enable these in every free-form text-entry field in apps you create—especially if you have comment forms or fields that are used to send a message. Yes, this feature works with the `RichEditBox` as well.

I'm happy to see the good-old `TextBox` here in WinRT. Not only does it work as expected, but it has features like spell check and autocorrect, which have finally made it into the core platform.

This `TextBox` was bound to model properties surfaced through the viewmodel. Let's look at another example where the `TextBox` is bound to properties of another control.

Like twins, they're different on the inside

The controls you use in XAML are based on the same specifications and same underlying animation libraries as those used in the OS and in HTML, but they're not the same physical controls. Each platform reimplements the controls to fit its own control model. WinJS/HTML apps have CSS-friendly versions. VB/C# and C++ apps have XAML-friendly versions.

This is important to note only in cases where there's a bug or a slight difference in functionality from one implementation or the other. In addition, when other development tools target WinRT, they also implement their own native versions of the controls.

This may seem ill-advised, but it's the best way to allow each UI platform to maintain its own approach and strengths without having to design everything to the lowest common denominator.

9.4 *UI element binding using sliders*

Whenever possible, I try to ensure that my UI elements are bound to properties surfaced through the viewmodel. But as a UI designer, there may be times when you want to do something purely for the UI's sake, without involving the viewmodel. This isn't just for dummying up the interface but also for real production work.

Figure 9.10 shows the part of the UI we'll be working with in this section. It includes two static `TextBlock` elements with labels, two `Slider` controls with ranges, min and max values and step values set, and two `TextBox` controls that can be used to display or set the values for the sliders.

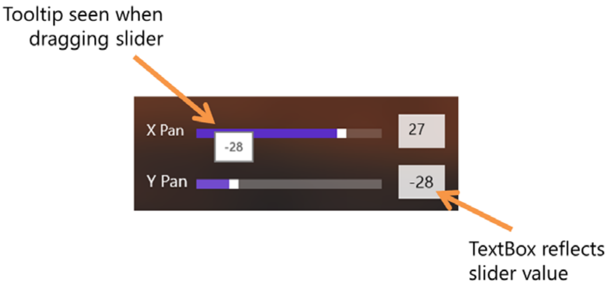

Tooltip seen when dragging slider
X Pan
-28
27
Y Pan
-28
TextBox reflects slider value

Figure 9.10 The X Pan and Y Pan UI elements. The tooltip shown is for the X Pan slider.

When working with the slider, you have a lot of control over how the data is spread across the control. First, you have the `Minimum` and `Maximum` properties, which set the values at either end of the slider. Next, there's the `SnapsTo` property, which lets you control whether intermediate values are allowed (good for smooth transitions if you need them) or values are snapped to the nearest interval. Finally, you have the `StepFrequency`, which lets you control the size of the steps across the control. It is unfortunately named, because it's more of a step value or interval than a frequency. Maybe I've been spending too much time around my oscilloscope lately.

When it comes to binding, you'll use the `Value` property of the `Slider`. This is a floating point value, so you can support some pretty fine-grained steps should you desire.

The next listing shows how to use UI element binding to bind together a `TextBox` and a `Slider`. Place this code in the area marked as reserved for the panning UI sliders.

Listing 9.14 UI Element binding in XAML linking a `TextBox` and `Slider`

```
<TextBlock Text="X Pan" VerticalAlignment="Center"
           FontSize="18" Grid.Column="0" Grid.Row="0" />
<Slider x:Name="XPosition
        Grid.Column="1" Grid.Row="0"
        Margin="10" Height="40"
        Minimum="-45" Maximum="45"
        SnapsTo="StepValues"
        StepFrequency="1"
        Value="0" />

<TextBox x:Name="XPositionEntry"
         Text="{Binding Value,ElementName=XPosition, Mode=TwoWay}"
         FontSize="18" Margin="10"
         Grid.Column="2"  Grid.Row="0"/>

<TextBlock Text="Y Pan" VerticalAlignment="Center"
           FontSize="20" Grid.Column="0" Grid.Row="1" />
<Slider x:Name="YPosition"
        Grid.Column="1" Grid.Row="1"
        Margin="10" Height="40"
        Minimum="-45" Maximum="45"
        SnapsTo="StepValues"
        StepFrequency="1"
        Value="0" />
<TextBox x:Name="YPositionEntry"
         Text="{Binding Value,ElementName=YPosition, Mode=TwoWay}"
         FontSize="20" Margin="10"
         Grid.Column="2" Grid.Row="1"/>
```

Slider current value

Slider snapping and steps

TextBox

Slider min and max values

UI element binding

Slider snapping and steps

Slider current value

This example uses element binding from within the `TextBox`. If you move the slider, the `TextBox` is updated. Because the binding is `TwoWay`, you can also type into the `TextBox` to set the value used by the slider.

I do like element binding in some pure UI scenarios (like binding a `TextBlock` to the number of characters entered in a `TextBox`), but you give up a bit of control by using it. Now let's look at the same binding, but this time it's handled through the viewmodel. This is the version we'll stick with for this chapter. The following listing delivers the XAML.

Listing 9.15 Accomplishing the same effect using the viewmodel

```
<Slider x:Name="XPosition"
        Value="{Binding SelectedCamera.XPosition, Mode=TwoWay}"
        ... />
<TextBox x:Name="XPositionEntry"
         Text="{Binding SelectedCamera.XPosition, Mode=TwoWay}"
         ... />


<Slider x:Name="YPosition"
        Value="{Binding SelectedCamera.YPosition, Mode=TwoWay}"
        ... />

<TextBox x:Name="YPositionEntry"
         Text="{Binding SelectedCamera.YPosition, Mode=TwoWay}"
         ... />
```

Bound to viewmodel

Most of the listing stays identical as before, so it shows only the changed lines and just enough around them for context. You can see that in each case the values are now bound to properties off of the viewmodel.

This relies on a `SelectedCamera` property on the viewmodel. The code for this is shown here.

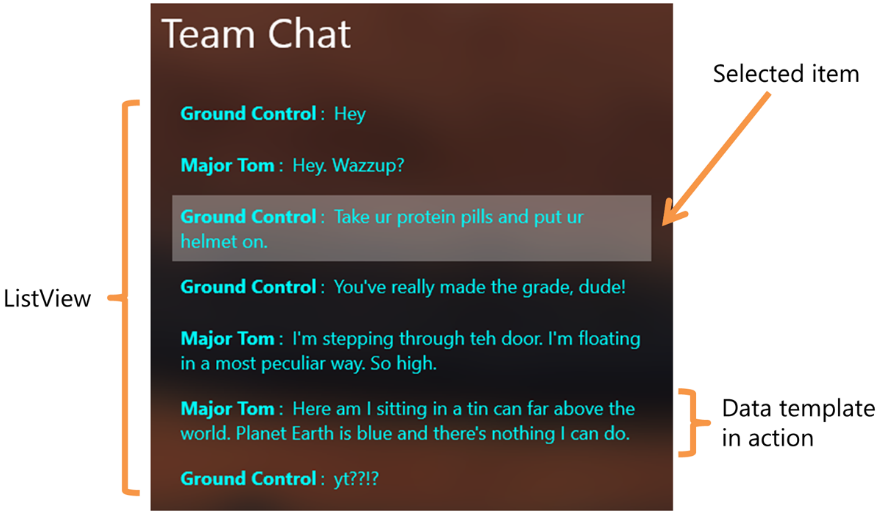
Listing 9.16 `MainViewModel SelectedCamera` property

```
private CameraViewModel _selectedCamera;
public CameraViewModel SelectedCamera
{
  get { return _selectedCamera; }
  set { Set<CameraViewModel>(
           () => SelectedCamera, ref _selectedCamera, value); }
}
```

You can see how this adds additional complexity to the viewmodel. But you may need this to be able to set the selected property outside the UI or to trigger other actions based on it changing.

UI element binding is, for the most part, a convenience. I don't recommend being dogmatic about whether you should allow this type of binding in your organization but to instead make decisions scenario by scenario. If there is no downstream effect or interest, binding UI elements together can be a time-saver and can simplify your UI model.

Let's look at something I really do recommend, and that's the approach for working with lists in the viewmodel and view.



Team Chat

Ground Control : Hey

Major Tom : Hey. Wazzup?

Ground Control : Take ur protein pills and put ur helmet on.

Ground Control : You've really made the grade, dude!

Major Tom : I'm stepping through teh door. I'm floating in a most peculiar way. So high.

Major Tom : Here am I sitting in a tin can far above the world. Planet Earth is blue and there's nothing I can do.

Ground Control : yt??!?

Figure 9.11 The list of chat messages used in this app.

9.5 Working with lists

Most consumption and many creation apps work with lists of data. Maybe it's a collection of photographs, a list of files, or, as we have here, a stream of messages from a chat service. XAML and binding work well with lists. In addition to the great support for the data side of lists already present in C#/XAML, Windows 8 adds in several new controls designed specifically for working with large lists. In this section, we'll briefly look at one of them, the `ListView`.

Figure 9.11 shows the list we're working with in our app. The data is displayed in a `ListView` using a data template.

List management begins with the source of the data. The most commonly used collection is the `ObservableCollection` because it has built-in support for binding and notification. When it comes to displaying the data, you'll use an `ItemsControl`, typically a `ListView` in Windows 8. Finally, to format all that data, you'll use a really cool feature called a data template. Let's look at each of those in turn.

9.5.1 Observable collections

If you need to surface a list of data to the view, you can't go wrong with an `ObservableCollection`. The `ObservableCollection` implements the `INotifyCollectionChanged` interface, which is for collections what `INotifyPropertyChanged` is for individual properties. As you add items to and remove items from the collection, the binding system is alerted to the changes via the events defined in this interface.

Listing 9.17 shows how to load the data from the data service into an `ObservableCollection` in the `MainViewModel`.

Listing 9.17 `MainViewModel` updates for loading the list of chat messages

```
private ObservableCollection<ChatMessage> _chatMessages =
    new ObservableCollection<ChatMessage>();
public ObservableCollection<ChatMessage> ChatMessages
{
  get { return _chatMessages; }
}

public void LoadMessages()
{
  var messages = ChatDataService.Current.GetMessages();

  foreach (var message in messages)
  {
    if (!ChatMessages.Contains(message))
      ChatMessages.Add(message);
  }
}
```

Binding source

Get messages

Add any new messages

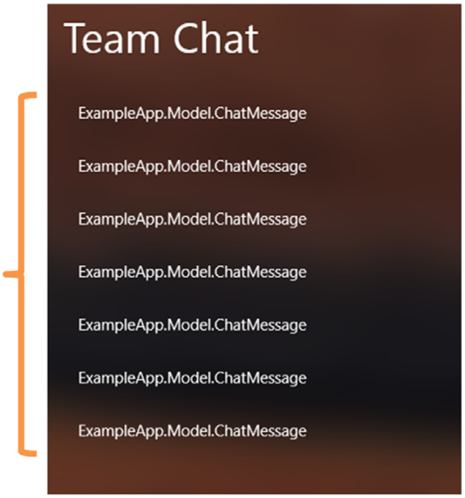
As the messages are added to the collection, the collection change events will fire (one per action; the actions aren't bundled), and the UI controls will update themselves. This is all completely transparent to you when you use an `ItemsControl`.

TIP Sometimes the source data is in a list, array, or collection, but you want to bind controls to the individual items, not to the list as a whole. To accomplish this, use the syntax `{ Binding PropertyName[indexOrKey] }` where `indexOrKey` is the string key for the dictionary (no quotes) or the numeric index. If the item stored at that index supports change notification through `INotifyPropertyChanged`, that will work as you expect.

9.5.2 Items controls

If you want to display more than one of something, you want an items control. Items controls are so named because they inherit from a common `ItemsControl` base class. This class provides the `ItemsSource` property, as well as support for templating of the items and for replacing the main items layout panel. It's these capabilities that make the items controls so flexible: You could have them use a `WrapGrid` or even something like the `OrbitPanel` I covered in chapter 7.

NOTE Not all items controls are equal, and not all items panels are swappable. For example, the new Windows 8 items controls all require the items panel to support virtualization. Conversely, the `ListBox` doesn't support virtualizing the same way the new controls do, so it can't use panels such as the `WrapGrid`. It takes some getting used to, because the errors aren't always obvious in their underlying cause.


Team Chat
ExampleApp.Model.ChatMessage
ExampleApp.Model.ChatMessage
ExampleApp.Model.ChatMessage
ExampleApp.Model.ChatMessage
ExampleApp.Model.ChatMessage
ExampleApp.Model.ChatMessage
ExampleApp.Model.ChatMessage

It looks like this because the default representation is the result of calling ToString() on the object.

Figure 9.12 What happens when you bind complex objects to a `ListView` but don't use a data template. The representation for each item is whatever it returns from `ToString`.

For years, the canonical items control was the `ListBox`. Windows 8 XAML still supports the `ListBox`, but the control has taken a backseat to those touch-oriented, animated, and virtualized heavyweights: the `ListView`, `FlipView`, and `GridView`. The most direct replacement for the `ListBox` is the new `ListView` control. The following listing shows how we're using that control in this UI. Place this code in MainPage.xaml in the space reserved for the message list.

Listing 9.18 Initial XAML for the chat list

```
<ListView x:Name="MessageList"
          ItemsSource="{Binding ChatMessages}"
          Grid.Column="0"
          Grid.ColumnSpan="2"
          Grid.Row="1"
          Margin="10" />
```

Collection binding

If you run the app now, you'll see data in the `ListView`, but the data will simply be a bunch of class names, as shown in figure 9.12. Although this does show you that binding is working, this isn't what you want to display. To get the data to display something meaningful, you need to use a data template.

9.5.3 Data templates

A `DataTemplate` is a chunk of XAML that's repeated for each item bound to an items control. Technically, you can use a `DataTemplate` any place where you use a `ContentControl` (through the `ContentTemplate` property), but it's by far used most often with `ItemsControl`-derived controls.

Inside the `DataTemplate`, you include any elements you want, inside a single parent. The elements can use data binding to display the data. The `ItemsControl` helpfully sets the data context for the template to the item being templated, so you don't have to worry about figuring out which position in the collection you're in or anything like that.

The next listing has the template version of the `ListView` control. The binding and layout have remained identical, but I added the `ItemTemplate`/`DataTemplate` property and formatted the data using a `TextBlock` with some `Run` elements.

Listing 9.19 Final XAML for the chat list

```
<ListView x:Name="MessageList"
          ItemsSource="{Binding ChatMessages}"
          Grid.Column="0"
          Grid.ColumnSpan="2"
          Grid.Row="1"
          Margin="10">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Grid>
        <TextBlock TextWrapping="Wrap" Margin="5"
                   Foreground="Aqua">
          <Run Text="{Binding From}"
               FontSize="18"
               FontWeight="Bold"/>
          <Run Text=": "
               FontSize="18"/>
          <Run Text="{Binding Message}"
               FontSize="18" />
        </TextBlock>
      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Collection binding

Binding in runs in template

Data template

When you run the app now, you'll see the nicely formatted list of chat messages, as shown at the very start of this section. Feel free to play around with the presentation within the template: You could add images, other controls, or anything you want.

Items controls are an essential part of the XAML platform. If you want to display any sort of list of repeating items, they're just what you need. If you want to display the data in a nicely formatted way, they support data templates. Data templates make great use of binding and data context to make it simple for you to display the right thing at the right position in the list.

So far, everything you've done has been working with data. What about cases when you want the user to actually do something, like click a button. What controls are available and how does that fit into our binding + MVVM story?

9.6 Making things happen with buttons and commands

When you want the user to be able to initiate an action, like sending a message, you can't beat a button. Buttons, including radio buttons and check boxes, all inherit from the `ButtonBase` control, which itself inherits from `ContentControl`. By being a

`ContentControl`, buttons can display any XAML for their interface—it doesn't have to be text or a simple image; anything you can represent in XAML can be the interface. By inheriting from `ButtonBase`, they all get the `Click` event, as well as command support.

Commands are an MVVM-friendly alternative to events. Rather than set up an event handler in the code-behind and execute code that way, you provide a command on your viewmodel and bind the button's `Command` property to that. This cuts the code-behind out completely and makes the code cleaner.

In this section, we'll look at several important buttons. First is the basic `Button` control, the one that you'd normally think of as a button. As part of that, you'll wire up a command with the viewmodel so that you can start adding chat messages to the list. Next, we'll look at the `HyperlinkButton`, which is useful for launching web pages. Then, we'll look at things you may never have thought of as buttons: `CheckBox` and `RadioButton` controls. Those inherit from a special button called the `ToggleButton`. Because we have to do a little something special to massage data, we'll wrap up this section with a hand-rolled value converter.

Let's start by providing some real interactivity with the UI. Let's make the Send button work.

9.6.1 Button and commands

In a nutshell, a button is a control that, when you tap or click it, raises a discrete `Click` event and calls an appropriate click command.

Given that the general pattern has remained the same since I started with VB3 in the early '90s, I'm going to make the assumption that you can double-click a button on the designer and figure out how the `Click` event works. So, instead, I'm going to jump right into using Laurent's `RelayCommand` to wire the button to the viewmodel rather than put code in the code-behind.

The `RelayCommand` and related functionality in the viewmodel are shown next. Place this code inside the `MainViewModel` class.

Listing 9.20 Surfacing a command from the viewmodel

```
public RelayCommand PostNewMessageCommand
{
  get; private set;
}
```

Command to bind to

Listing to PropertyChanged (replace existing method)

```
void NewMessage_PropertyChanged(object sender,
                                PropertyChangedEventArgs e)
{
  if (e.PropertyName == "Message")
  {
    PostNewMessageCommand.RaiseCanExecuteChanged();
  }
}
```

Post new message

```
public void PostNewMessage()
{
    NewMessage.From = "Ground Control";

    ChatMessages.Add(NewMessage);

    CreateNewMessage();
}

public bool CanPostNewMessage()
{
    return NewMessage != null &&
           !string.IsNullOrEmpty(NewMessage.Message);
}
```

For enabling the button

This listing includes the code to implement the `RelayCommand` `PostNewMessageCommand`. This command is what's used from the button on the UI.

Also in the listing, you can see where you're again calling `CreateNewMessage` and why the cleanup code in that function is necessary. The actual `ChatMessage` instance gets added to the collection; in order to make the text entry work with a new instance, you need to remove the old event handlers (prevent leaks), create a new instance, and rewire the event handler. Many people forget to unwire their event handlers—the main cause of memory leaks in many .NET apps.

I created a `CanPostNewMessage` function. In fact, that function is the reason I'm capturing the `PropertyChanged` event from the `NewMessage` instance. How is that used? Before we get to that, let's add a little code that will help with the explanation. After the `CreateNewMessage` call in the `MainViewModel` constructor, add the command setup code shown here.

Listing 9.21 Initializing the `PostNewMessageCommand`

```
public MainViewModel()
{
    ...

    PostNewMessageCommand = new RelayCommand(
        () => PostNewMessage(),
        () => CanPostNewMessage());
}
```

Action to perform

Can action be performed?

This code sends two lambda expressions to the `RelayCommand` constructor. The first is the action the command will perform; the second is the code to use to check to see if the command can be executed. This not only works as a gate to the function but more importantly is what enables or disables the button that's bound to the command. Yes, command-aware controls use this code to control their state, without any additional UI logic on your part. That's really helpful, but it did require the additional viewmodel code.

Because the command doesn't automatically know when to reevaluate `CanPostNewMessage`, you have to tell it when it should. This is done by calling the command's `RaiseCanExecuteChanged` method, called from the viewmodel. You call that when the property change notification is received. In this way, you ensure the button is enabled if the viewmodel contains text in the `NewMessage` instance.

Finally, to wire this all up, it takes only a single binding statement in the `Button`'s markup. The following listing has everything you need.

Listing 9.22 The `Button` XAML bound to the viewmodel

```
<Button x:Name="SendMessage"
        Command="{Binding PostNewMessageCommand}"
        VerticalAlignment="Top"
        Margin="5 0 0 0"
        Content="Send"
        Grid.Column="1"
        Grid.Row="3" />
```

Command binding

Just as with data-binding statements, command-binding statements are enclosed in curly braces with the `Binding` keyword. In this case, the property to be bound to must implement `ICommand`, as the `RelayCommand` does. You may also add an optional parameter using the `CommandParameter` property; this can be useful, for example, when you want several controls to share the same command but pass in distinguishing values.

One thing I didn't show you in this discussion of the button is the use of events and code-behind. That's because, if you decide to go with commands, you won't need click events for your buttons. Sure, if you want to whip something up quickly, events will get you there. For anything that has the potential to be even slightly complex (or that may have the UI turned over to a design team), definitely consider using commands.

When you think of button controls, the first one to come to mind is the standard button. But it's also common to find the `HyperlinkButton` in modern apps.

9.6.2 HyperlinkButton

The `HyperlinkButton` is a special kind of button that's allowed to launch a URL in the browser (or other app that has registered for the specified protocol in the URL). If you provide a `NavigateUri` value, clicking the button will launch the browser and navigate to that page. This all happens as another app, not inside the running app. Your app will suspend as appropriate, and control will be transferred to the browser.

If you'd rather not launch a browser, you can also handle events and commands on the button just as you would a regular button. But unless you have a specific reason for doing so, I encourage you not to use the `HyperlinkButton` this way. The next listing shows the typical use for a `HyperlinkButton`. (This markup already exists in our app.)

Listing 9.23 The `HyperlinkButton` markup as used in this app

```
<HyperlinkButton x:Name="ViewOnlinePage"
                 HorizontalAlignment="Left"
                 VerticalAlignment="Bottom"
                 Foreground="Orange"
                 Margin="10"
                 FontSize="18"
                 NavigateUri="http://www.nasa.gov"
                 Content="View NASA Homepage" />
```

NavigateUri

Content

The `Content` property defines what shows up for the UI for this button. As with any other content control, you can set this to anything you want: a grid with images and text, plain text, shapes, and more.

We're setting it in XAML, but you can use code or binding to set the `NavigateUri` property. In this case, when the button is clicked, it opens the browser and navigates to the NASA site.

Custom protocols

Windows 8 apps can register themselves as protocol handlers. The primary intent is to support apps that serve as VOIP handlers and similar, but you can register any protocol you want and become the handler for that.

Once registered, `HyperlinkButton` controls that reference that protocol will launch your app. Similarly, if the URL is entered into Internet Explorer or clicked in email or other apps, your app will be activated. This can be extremely useful for sending deep links to content in your app (like a specific customer number, for example).

You can find out more about creating and registering a custom protocol handler app on MSDN: http://bit.ly/Win8CustomProtocolHandler.

Both of the previous buttons, the `HyperlinkButton` and the `Button` itself, invoke an action when clicked in normal user interaction. Neither maintains any state like a toggle button would, but are instead click-and-release-type buttons. The next two types of buttons are both inherited from `ToggleButton`, which lets them maintain a checked state once clicked.

9.6.3 RadioButton and CheckBox

Further moving away from core button functionality, we have two controls that aren't usually used to perform an action but rather to set an option: the `RadioButton` and `CheckBox`. You're likely familiar with these controls: A `RadioButton` allows the user to select only one instance of the control within a group, and a `CheckBox` allows you to select any number of instances. `RadioButton` controls are typically round (or diamond-shaped if you want to go all Motif on me), and `CheckBox` controls are typically square.

VIEWMODEL SUPPORT FOR RADIO BUTTONS

There are lots of ways to handle binding with `RadioButton` controls, but none of them are good. A simple search on `StackOverflow` for `RadioButton` MVVM or similar will show you the lengths people go to try to get binding to work correctly with radio buttons.

The reason binding with radio buttons is so difficult is that binding is inherently a 1:1 relationship, whereas radio buttons are a 1:many relationship with some data. What you really need is something like the `RadioButtonList` from ASP.NET or a port of one from the various WPF-compatible versions. If your UI is likely to have a lot of radio button controls, I encourage you to use that.

The WPF versions rely on a few things simply not supported in WinRT XAML. One in particular, setting the radio button controls as not focusable, just isn't there. I have yet to see a good port of the `RadioButtonList`, and my own attempts haven't yielded anything I'd want to put my name on.

What I've decided to show you here is how you can bind the `IsChecked` property of the radio button to an `IsSelected` property in the `CameraViewModel` class. If this class were a regular model class, adding a UI support property would give you more than a little heartburn. As it is, this is a viewmodel class, so there are no issues. The next listing shows the additional property.

Listing 9.24 Update to the `CameraViewModel` class

```
private bool _isSelected = false;
public bool IsSelected
{
    get { return _isSelected; }
    set { Set<bool>(() => IsSelected, ref _isSelected, value); }
}
```

The downside of this approach is that it requires manually updating the linked `SelectedCamera` property. This can be handled in the viewmodel by listening for the `PropertyChanged` event on the `CameraViewModel` instances and then setting the `SelectedCamera` as appropriate. Here's how.

Listing 9.25 `RadioButton` support code in `MainViewModel`

```
private CameraViewModel _selectedCamera;
public CameraViewModel SelectedCamera
{
  get { return _selectedCamera; }
  private set
  {
    Set<CameraViewModel>(() => SelectedCamera, ref _selectedCamera, value);
  }
}

void CameraViewModel_PropertyChanged(object sender,
                                     PropertyChangedEventArgs e)
{
  var cvm = sender as CameraViewModel;

  if (e.PropertyName == "IsSelected" && cvm.IsSelected)
    SelectedCamera = cvm;
}
```

Set selected camera

I'm not a fan of this type of plumbing code. But because it's encapsulated in the viewmodel, it's both transparent to the UI and is testable.

Whenever the radio button is checked, the appropriate `CameraViewModel` instance's `IsSelected` property is set to `true` (the previous one is automatically set to `false`), and the `SelectedCamera` property is updated. The `CameraViewModel_PropertyChanged` event handler is responsible for synchronizing the `SelectedCamera` property with the camera, which has `IsSelected` set to `true`.

To make this work, you'll need to hook up the event handler after the `LoadImages` call in the constructor. You've added a lot to the constructor, so the following listing shows the complete version, including this addition.

Listing 9.26 Complete `MainViewModel` constructor

```
public MainViewModel()
{
  _cameras = new CameraViewModel[3];
  for (int i = 0; i < _cameras.Length; i++)
  {
    _cameras[i] = new CameraViewModel();
  }

  _cameras[0].Name = "Front camera";
  _cameras[1].Name = "Rear camera";
  _cameras[2].Name = "Horizon camera";

  LoadImages();

  _cameras[0].PropertyChanged += CameraViewModel_PropertyChanged;
  _cameras[0].IsSelected = true;

  CreateNewMessage();
  PostNewMessageCommand = new RelayCommand(
    () => PostNewMessage(),
    () => CanPostNewMessage());
}
```

Initialize cameras

Load images

Wire up event handler

Select initial camera

Support message posting

The constructor code now creates the cameras, initializes them with names, loads in the images, wires up the event handler for the radio buttons, and finally sets up the command support for creating a new message.

RADIOBUTTON BINDING AND MARKUP

Finally, we have the markup that makes use of all of this. The next listing includes the markup to add into the spot in the XAML reserved for the radio buttons.

Listing 9.27 RadioButton markup

```
<RadioButton Content="{Binding Cameras[0].Name}"
             GroupName="CameraControl"
             IsChecked="{Binding Cameras[0].IsSelected, Mode=TwoWay}"
             FontSize="18"/>
<RadioButton Content="{Binding Cameras[1].Name}"
             GroupName="CameraControl"
             IsChecked="{Binding Cameras[1].IsSelected, Mode=TwoWay}"
             FontSize="18"/>
<RadioButton Content="{Binding Cameras[2].Name}"
             GroupName="CameraControl"
             IsChecked="{Binding Cameras[2].IsSelected, Mode=TwoWay}"
             FontSize="18"/>
```

Binding

Although you're not doing anything with the selected camera, this approach is a common pattern you should know. It was important to show how to synchronize the radio buttons with the rest of the system so you can use this when creating your own apps.

Note also the use of indexed binding both for the radio button's content as well as for the `IsChecked` property.

WORKING WITH THE CHECKBOX

The `CheckBox` is almost identical to the `RadioButton` with one really important distinction: Button selection isn't mutually exclusive. Because of this, there's no need for all the crazy workarounds you need with the `RadioButton` instances.

The check boxes in this UI simply toggle the value of `IsEnabled` on the cameras. You'll use this to control the display of the image elements. The next listing shows the `CheckBox` side of the equation; put this markup in the MainPage.xaml spot reserved for the check boxes.

Listing 9.28 CheckBox markup

```
<CheckBox Content="Front Camera Enabled"
          IsChecked="{Binding FrontCamera.IsEnabled, Mode=TwoWay}"
          FontSize="18"/>
<CheckBox Content="Rear Camera Enabled"
          IsChecked="{Binding RearCamera.IsEnabled, Mode=TwoWay}"
          FontSize="18"/>
<CheckBox Content="Horizon Camera Enabled"
          IsChecked="{Binding HorizonCamera.IsEnabled, Mode=TwoWay}"
          FontSize="18"/>
```

TwoWay CheckBox binding

You use `TwoWay` binding so changes in code will be reflected in the UI and vice versa. Binding a Boolean property to a Boolean property is simple; nothing else is required.

You can see that working with `CheckBox` controls is generally quite a bit easier than working with `RadioButton` controls. If you're looking at a UI with a number of radio buttons, I encourage you to settle on a single pattern that works for you, even if it's not perfect, and use that in every place in the app.

Back to the last example; for the `CheckBox` side of this equation, no additional support code is needed. You'll need to add a little bit to the `Image` binding to properly convert the `IsEnabled` property into something you can use. That's the final topic for this chapter.

9.7 Converting data with value converters

Sometimes, the data that's the binding source simply isn't compatible with the binding target. In many cases, WinRT will handle the conversion for you. There are many cases, however, when this simply isn't possible. One such case is converting a Boolean value to a `Visibility` enumeration.

Converting `true` to `Visibility.Visible` and `false` to `Visibility.Collapsed` is a very common task. For that, you'll use a special component called a *value converter*. A value converter is code you can write yourself and then use from markup to intercept the data being used in binding.

A value converter implements the `IValueConverter` interface, which has two functions: `Convert` and `ConvertBack`. Each time the binding source is updated, your `Convert` function will be called. Each time the binding target is updated, your `ConvertBack` function will be called.

Create a new folder named Converters. In that, create a new class named `BooleanToVisbilityConverter` and paste in the following code.

Listing 9.29 The `BooleanToVisibilityConverter`

```
using System;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Data;

namespace ExampleApp.Converters
{
  public sealed class BooleanToVisibilityConverter : IValueConverter
  {
    public object Convert(object value, Type targetType,
                          object parameter, string language)
    {
      return (value is bool && (bool)value) ?
                 Visibility.Visible : Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, string language)
    {
      return value is Visibility &&
                   (Visibility)value == Visibility.Visible;
    }
  }
}
```

Convert source (bool) to target (Visibility)

Convert target (Visibility) back to source (bool)

This converter actually comes in the project file for the richer XAML project templates, like the grid and split views. It's not included in the blank template, nor is it included in the MVVM Light templates. It's really useful, though. Use this when you want to toggle the visibility of something on the UI using a Boolean value in your viewmodel.

To use the converter, you must surface it as a resource. Typically, you'd put this in App.xaml to make it available to the entire app. But you're doing everything else on the page, so you'll put it there. The following listing shows two things you need to add: the converters namespace and the resources themselves.

Listing 9.30 Surfacing the converter as a page resource

```
<Page x:Class="ExampleApp.MainPage"
  ...
  xmlns:converters="using:ExampleApp.Converters"
  ...>

  <Page.Resources>
```

Namespace in Page tag

```
    <converters:BooleanToVisibilityConverter x:Key="BTVC" />
  </Page.Resources>
...
```

Resource declaration in Resources

If you still have the resources group from the template, put the BTVC (named as such just to make it easier to deal with in print) declaration inside the `ResourceDictionary` block, or replace the entire block with this `Page.Resources` block.

Finally, once the converter is there as a page resource, all you need to do is reference it from within the binding statement on the `Image` elements. Here's the updated `Image` markup.

Listing 9.31 Final `Image` binding markup

```
<Image x:Name="FrontCameraOutput"
  Source="{Binding Cameras[0].Image}"
  Visibility=
    "{Binding Cameras[0].IsEnabled, Converter={StaticResource BTVC}}"
  Margin="10"
  Stretch="Uniform"/>

<Image x:Name="RearCameraOutput"
  Source="{Binding Cameras[1].Image}"
  Visibility=
    "{Binding Cameras[1].IsEnabled, Converter={StaticResource BTVC}}"
  Margin="10"
  Stretch="Uniform"/>

<Image x:Name="HorizonCameraOutput"
  Source="{Binding Cameras[2].Image}"
  Visibility=
    "{Binding Cameras[2].IsEnabled, Converter={StaticResource BTVC}}"
  Margin="10"
  Stretch="Uniform"/>
```

Converter in use

The value converter handles, as the name implies, the conversions of values going between the binding source and the target. In this case, when the camera's `IsEnabled` property is changed, it will first pass through the converter's method before the `Image` elements see it.

Value converters are useful for many different types of operations. Often, if the binding model doesn't support a particular binding type you need, you can create something analogous using a value converter.

9.8 Summary

WinRT XAML supports most of the controls you'd normally expect to find in the UI layer of a development platform. In addition to the static text and shapes covered in other chapters, you also have multiline text entry with spell checking and autocorrect, check boxes and radio buttons for selecting options, sliders for setting values, regular buttons for invoking actions, and even hyperlink buttons for launching something based on a URI.

XAML also brings to the table something that's not universally shared with UI platforms: strong binding support. You can bind controls and elements to each other, like the `Slider` and `TextBox` shown in this chapter. More commonly used, you can bind controls to data elements and control actions to commands.

When binding controls to data, the most commonly used pattern is the Model-View-ViewModel pattern. This pattern encourages separation of the logic from the presentation, while using binding to communicate between the layers. Data uses regular binding; commands use command binding. The end result is easily separated layers that can be cleanly developed and tested. If you need to massage the data, you can do that in the viewmodel or through a special type of class called a value converter.

I've found it easiest to use a third-party MVVM toolkit, like MVVM Light when implementing the MVVM pattern. There are many MVVM toolkits out there; I encourage you to investigate several of them before settling on a single one.

There are more controls beyond the basics covered here, but they're easily covered in the context of other topics. In the next chapter, we'll expand on the knowledge gained here, and you'll learn about the newer items controls available in Windows 8: the `GridView`, `ListView`, and `FlipView`.

10

View controls, Semantic Zoom, and navigation

This chapter covers

- The `ListView`, `GridView`, and `FlipView` controls
- Grouping data for display
- Page navigation
- Semantic Zoom

In previous chapters, you've learned how important grid layout is to the UI for Windows 8 apps. This applies not only to things the designer places on screen at design time, like buttons and captions, but also to dynamically loaded content.

After you log into a Windows 8 machine, the first thing you see on screen is a grid of tiles: the Start page. This page uses an implementation of the `GridView` control, as well as Semantic Zoom (pinch the display to see that in action) to assist in organizing and grouping the tiles for apps installed on the PC. Look at other apps on the system; with the exceptions of games and highly specialized UIs (like painting and music creation), chances are you'll see items arranged in grids or lists in

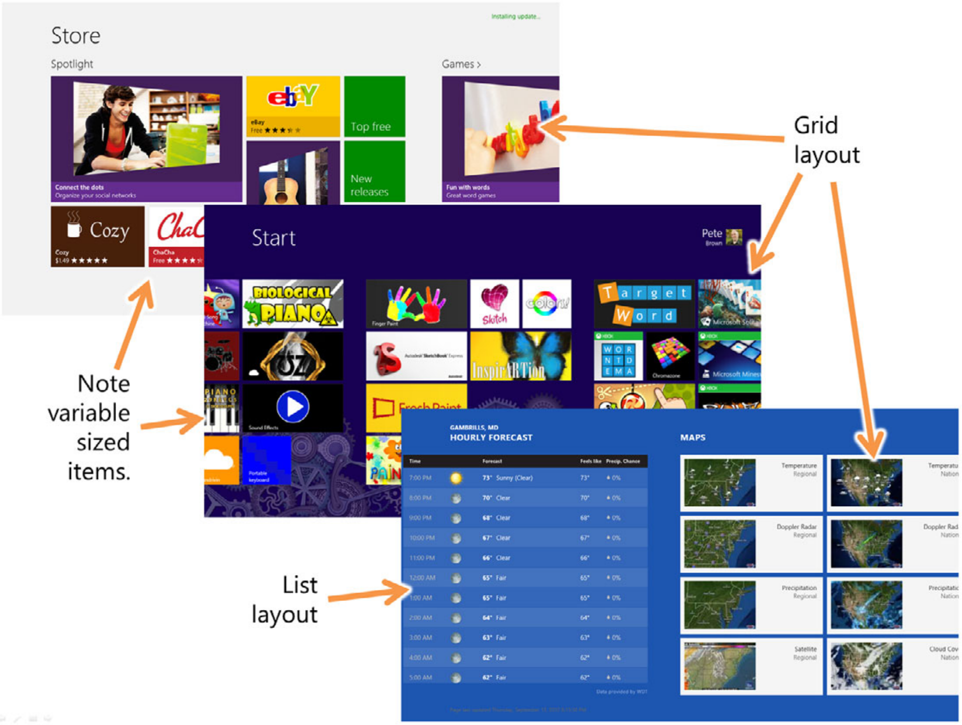


Figure 10.1 Examples of lists and grid layouts in apps that come with the PC

most of the apps. Figure 10.1 shows just a few examples of list and grid layouts within the default apps and the start page.

In the past, new UI controls were always introduced into the OS and key products like Microsoft Office long before we had the ability to use them in our own apps. In Windows 8, these controls are so central to the user experience that they've been included in each UI stack from the very start. If you want to make an app that looks like those in the Windows Store, complete with headers that may be used for navigation, you can do that; Microsoft used the same control libraries available to app developers.

For those of you who have been programming for a while, the `ListView`, `GridView`, and `FlipView` controls are brand-new controls introduced with Windows 8. Each of these controls displays collections of data using templates to control layout and representation. Each of these controls relies very heavily on binding to get the data items in and to format each individual data item for display.

> **NOTE** Although the `ListView` control is named the same, it bears no resemblance to the classic `ListView` control used in Windows desktop apps.

In the previous chapter, you created an app using MVVM Light. You're going to do the same in this chapter. If you skipped over the MVVM coverage, you may want to brush

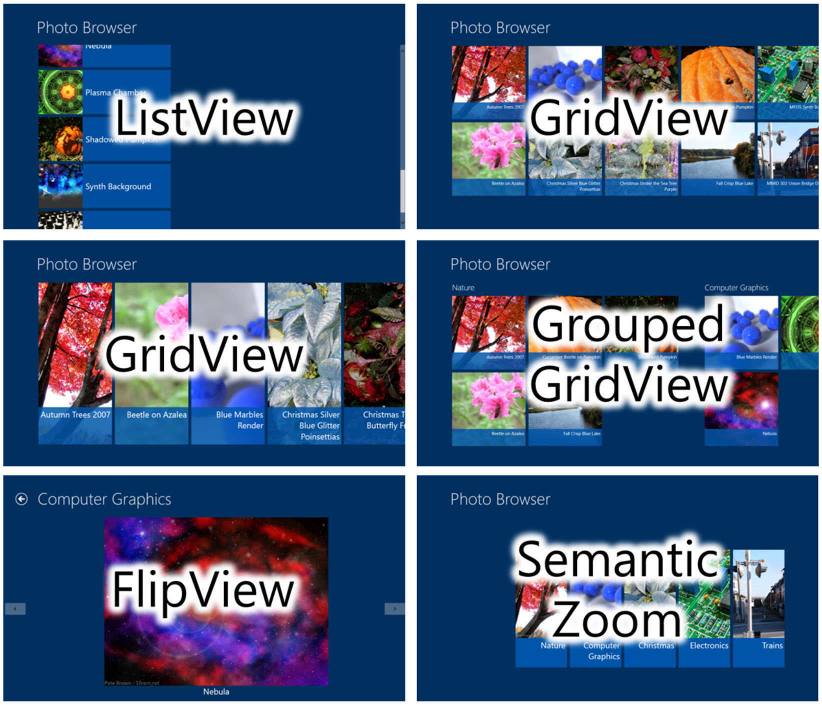


Photo Browser
ListView
Plasma Chamber
Shadowed Pumpkin
Synth Background
Photo Browser
GridView
Photo Browser
GridView
Autumn Trees 2007
Beetle on Azalea
Blue Marbles Render
Christmas Silver Blue Glitter Poinsettias
Photo Browser
Nature
Computer Graphics
Grouped GridView
Computer Graphics
FlipView
Nebula
Photo Browser
Semantic Zoom
Nature
Computer Graphics
Christmas
Electronics
Trains

Figure 10.2 The PhotoBrowser app showing the six areas we'll concentrate on in this chapter

up on that now, because you'll use it in the app you create here and expand throughout this chapter and several following chapters.

The demonstration app for this chapter will be a simple photo browser. It will, using several different list controls, show you the images loaded from a dummy service. This chapter concentrates on the view controls, Semantic Zoom, and navigation, so the app will use each of those features. In future chapters, you'll expand its capabilities to show how to implement other Windows 8 features.

Figure 10.2 shows the six different incarnations of the app. First, you'll create a `ListView` to show how to work with (typically) vertical lists of data. The `ListView` has been shown in several other places in this book and is not as feature rich as the `GridView`, so we won't dwell too long on it here. Next, you'll use two different item templates with the `GridView`. Then, you'll move into grouping with the `GridView`. Once you learn how to group, you'll modify the group headers to be links to a page that shows the details for the group. The details page will use a `FlipView` control to navigate between each image in that group. Finally, you'll implement Semantic Zoom using the `SemanticZoom` control, allowing you to take a high-level look at the groups.

10.1 PhotoBrowser demonstration app setup

Some concepts are best shown with real demonstration apps. The UI subjects covered in this chapter and the next two definitely fall into that category. In fact, they can all

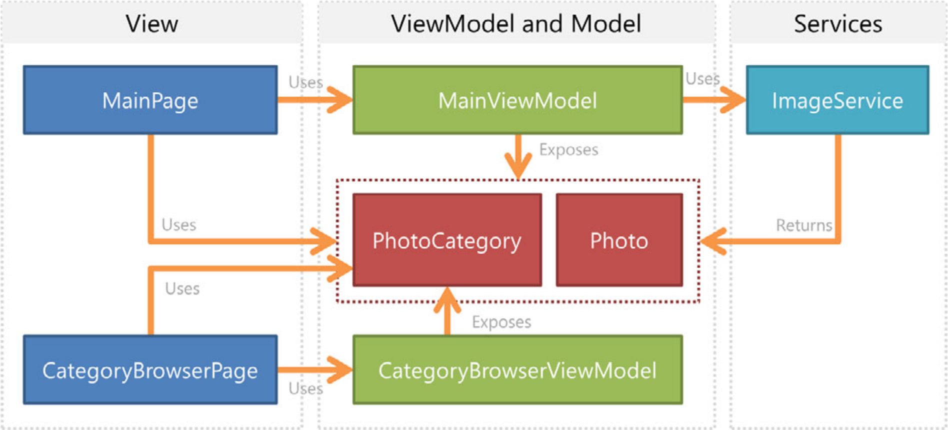


Figure 10.3 The overall architecture of the completed PhotoBrowser app. Some lines are simplified: The CategoryBrowserPage technically uses the `Photo` model object, but really it's using that only through the `PhotoCategory` object's `Photos` collection. Feel free to debate this distinction with the architects in your organization (they aren't busy anyway).

use the same demonstration app, gradually built over the course of the chapters. You'll also build on the MVVM knowledge from the previous chapter.

This section introduces the new sample app, so you have a little setup to do. Figure 10.3 shows the overall architecture of the app. By the end of this chapter, you'll have two different pages, each talking with its own viewmodel. The viewmodels will expose instances of `Photo` and `PhotoCategory` objects. They'll also use the `ImageService` class to load image information into the viewmodel.

In this section, you'll create an MVVM Light Windows Store app. Next, because we're going to defer the file IO topics to an upcoming chapter, you'll drag a few photos directly into the project and use them for the data to show. Then, you'll create the model, viewmodel, and service classes required to implement the MVVM pattern. We'll wrap up this section with the skeleton XAML and code-behind. The app built in this section will then be ready for you to use to explore the different view controls, navigation, and Semantic Zoom.

10.1.1 Creating the project

The first step is to create the project. Create a new MVVM Light Windows Store project. I named mine PhotoBrowser. To ensure the namespaces in the code listings in this chapter work as written, you'll want to name yours the same.

As before, you're not going to use much of what MVVM Light has in terms of viewmodel locator services, data services, and more. For those reasons, if you wish to use another MVVM toolkit, you should find the conversion pretty straightforward. What we'll use in this chapter is the `ViewModelBase` class. In subsequent chapters, you'll also make use of commanding. You may remove the design time data, locator classes, and resources as I have in the downloadable source code, but you'll need to remove the resource references from App.xaml and the pages if you do so.

Next, off the root of the project, create a folder named Pictures and drag the sample pictures into it. These are pictures I've taken over the years and have posted on my site as downloadable wallpaper. For simplicity, the photos you're loading are included in the project itself.

At this point, you should have a base MVVM app with one new additional folder, several pictures, and no real functionality. The app should compile without errors.

10.1.2 Creating the Photo model class

For this version of the app, you'll have a class that represents a photo, or picture in the system. The class includes an `ImageUri` property, which will be a `ms-appx:/` URI pointing to the image in the project. We covered this URI protocol in chapter 7. The `DisplayName` property is the image filename without the extension, just as the OS does when you use native image loading APIs. Finally, to support the grouping work you'll do later, the photo needs a `Category` property. This is simply a string name with a category we'll assign in code.

In the existing Model folder, create a new class named `Photo`. The code for this class is in the following listing.

Listing 10.1 The `Photo` model class

```
using System;

namespace PhotoBrowser.Model
{
  public class Photo
  {
    public Uri ImageUri { get; set; }
    public string DisplayName { get; set; }
    public string Category { get; set; }
  }
}
```

An ms-appx URI

Friendly name

Category name

The properties of the `Photo` class won't change once the class is created. For that reason, the properties don't need to raise the property changed event. Another, and arguably more correct, object-oriented design approach here would have been to make the properties read-only and have an initialization constructor that took the three properties as parameters. I'm not a big fan of constructors that do nothing more than initialize properties. But if you want to take that approach here, you'll have only one other place to change the code: the service class.

10.1.3 Loading pictures using a service class

MVVM apps often make use of service classes to handle integration with functionality that's external to the app. As I covered in the previous chapter, a *service* here doesn't mean a web service; it's simply a class that provides an interface to functionality or data. In the case of this app, the service will encapsulate the loading of the images.

Create a folder named Services, and in it create a new class named `ImageService`. The source for this class is shown in the following listing.

Listing 10.2 The `ImageService` class

```
using PhotoBrowser.Model;
using System;
using System.Collections.Generic;

namespace PhotoBrowser.Services
{
  public class ImageService
  {
    private static ImageService _current;
    public static ImageService Current
    {
      get
      {
        if (_current == null)
          _current = new ImageService();

        return _current;
      }
    }

    public IList<Photo> GetPhotos()
    {
      string[] fileNames = new string[]
      {
        "Autumn Trees 2007.jpg", "Nature",
        "Beetle on Azalea.jpg", "Nature",
        "Blue Marbles Render.jpg", "Computer Graphics",
        "Christmas Silver Blue Glitter Poinsettias.png", "Christmas",
        "Christmas Tree Butterfly Fruit.jpg", "Christmas",
        "Christmas Under the Sea Tree Purple.jpg", "Christmas",
        "Cucumber Beetle on Pumpkin.jpg", "Nature",
        "Fall Crisp Blue Lake.png", "Nature",
        "MFOS Synth Board 2.png", "Electronics",
        "MMID 302 Union Bridge Dec 15, 2004.png", "Trains",
        "Nebula.jpg", "Computer Graphics",
        "Plasma Chamber.jpg", "Computer Graphics",
        "Shadowed Pumpkin.jpg", "Nature",
        "Synth Background.jpg", "Electronics",
        "Synth Panel.jpg", "Electronics"
      };

      var photos = new List<Photo>();

      for (int i = 0; i < fileNames.Length; i +=2)
      {
        string name = fileNames[i];
        string category = fileNames[i + 1];

        photos.Add(
          new Photo()
          {
```

Singleton support

Image filenames and categories

Array iteration (notice stepping)

Photo creation

```
                ImageUri = new Uri("ms-appx:/Pictures/" + name),
                DisplayName = name.Substring(0, name.LastIndexOf('.')),
                Category = category
              });
        }

        return photos;
      }
    }
}
```

This class implements the singleton pattern, so that only one instance of it will be created in the app. The single function, `GetPhotos`, returns a list of photos located in the Pictures folder of the app.

The array is a convenience to reduce the lines of code in the listing. If you use different filenames, you'll need to update them in the array. Note also that you can have any number of different files listed in the array, as long as you follow each filename with its category.

If you created a property initialization constructor, you'll need to change the code inside the `for` loop where the photo instance is created.

Now that you have a way to get the photos from the project, you need to expose them to the rest of the app through the viewmodel.

10.1.4 Creating the MainViewModel

To start, the app will have only a single view and viewmodel. The `MainViewModel` will surface all of the data and functionality required by the MainPage. The `viewmodel` class itself was already created as part of the project template, so you don't need to add anything. Simply wipe out the contents of the file and replace it with what you see in the next listing.

Listing 10.3 `MainViewModel` class

```
using GalaSoft.MvvmLight;
using PhotoBrowser.Model;
using PhotoBrowser.Services;
using System.Collections.ObjectModel;
using System.Linq;

namespace PhotoBrowser.ViewModel
{
  public class MainViewModel : ViewModelBase
  {
    private ObservableCollection<Photo> _photos =
            new ObservableCollection<Photo>();
    public ObservableCollection<Photo> Photos
    {
      get { return _photos; }
    }

    public void LoadPhotos()
```

Collection of photos

Function to load photos

```
      {
        var photos = ImageService.Current.GetPhotos();

        Photos.Clear();
        foreach (var photo in photos)
          Photos.Add(photo);
      }
    }
  }
```

Call to service class

The `MainViewModel` class, like most viewmodel classes, is a place to hang data and abstract away calls to service classes. The `LoadPhotos` function calls the `GetPhotos` method of the `ImageService` class and iterates over the results, adding them to the `ObservableCollection`, which is used for binding in the UI.

10.1.5 Skeleton UI XAML and code-behind

You're getting closer to the UI now: You have the service to load the photos, the model class to hold the photo information, and the viewmodel class to expose that information to the view. All you need now is the view itself.

The project template comes with a MainPage.xaml file created by default. Keep the file, but replace its contents with those shown here.

Listing 10.4 MainPage XAML skeleton

```
<Page x:Class="PhotoBrowser.MainPage"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">

  <Page.Resources>
    <CollectionViewSource x:Key="PhotoSource"
                          IsSourceGrouped="False"
                          Source="{Binding Photos}"/>
  </Page.Resources>

  <Grid Background="#FF003060">
    <Grid.RowDefinitions>
      <RowDefinition Height="140"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Grid Grid.Row="0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="116"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>

      <TextBlock x:Name="pageTitle"
                 Text="Photo Browser"
                 Grid.Column="1"
                 IsHitTestVisible="false"
                 Style="{StaticResource PageHeaderTextStyle}"/>
```

Binding source

App title

```
    </Grid>

    <Grid Grid.Row="1">

      <!-- Grids and lists will go here -->

    </Grid>
  </Grid>
</Page>
```

Placeholder

A new element in this listing is the `CollectionViewSource`. This is a class we haven't previously discussed. The `CollectionViewSource` provides a way to surface data to elements on the page. It has built-in support for grouped data, which is essential to some of the things I'll show later in this chapter. Figure 10.4 shows the relationship between the on-page view controls (`ListView` and `GridView`), the `CollectionViewSource`, and the `MainViewModel` class.

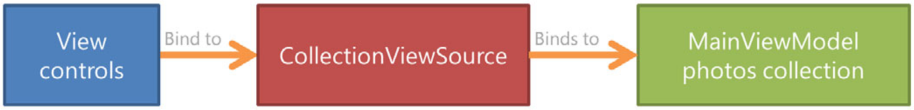


Figure 10.4 **The relationship between the `ListView` and `GridView` controls, the `CollectionViewSource`, and the viewmodel**

You can always use the `CollectionViewSource` as your data source when binding to collections. In most cases, when using MVVM, it doesn't provide anything worth the additional abstraction. But it does provide a level of synchronization of the current item between controls (important when using Semantic Zoom), as well as support for grouped items (important primarily for the `GridView`).

The code-behind for this app creates the connection with the viewmodel and then, when the page is navigated to, calls the function to load the photos. As in the previous chapter, you wire up the viewmodel and data context in the code-behind. You're welcome to use a viewmodel locator class if you prefer. The next listing shows my version; replace the code-behind in MainPage.xaml.cs with the listed source code.

Listing 10.5 **Code-behind to wire up the `MainViewModel`**

```
using PhotoBrowser.ViewModel;
using System.Diagnostics;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace PhotoBrowser
{
  public sealed partial class MainPage : Page
  {
    private MainViewModel _vm = new MainViewModel();
```

Create viewmodel

```
    public MainPage()
    {
      this.InitializeComponent();

      DataContext = _vm;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
      _vm.LoadPhotos();
    }
  }
}
```

Set data context

Load photos

The code-behind does two simple things: It wires up the viewmodel as the `DataContext` for the page, and it loads the photos when the page is navigated to.

Everything so far has been plumbing to support viewing data in the `ListView` and `GridView` controls (and the `FlipView`, but that will require an additional page and a second viewmodel, as you'll see).

If you run the app at this point, you should see a dark-blue background with a white title. Although the data is loaded, there's nothing in the UI to display the data. Through the remainder of the chapter, you'll use different controls to display the lists of photos returned from the service and exposed via the viewmodel.

10.2 ListView and GridView

In Silverlight and WPF, the `ListBox` was the primary control used for displaying lists of data (also, sometimes the `ItemsControl` base class was used directly). In WinRT XAML, there are two primary controls you can use when you need to display data in some sort of list: the `ListView` and the `GridView`.

Both of these controls are optimized for touch-based interaction and UI virtualization. You'll find that in Modern UI apps, you'll rarely, if ever, use a regular `ListBox` control—it wasn't designed with the modern interaction and virtualization approaches in mind.

The new view controls themselves are common across all UI stacks in Windows. You'll find `ListView` and `GridView` equivalents in HTML, for example. But with the exception of the use of OS-supplied animations and touch support, the underlying implementations are completely different.

In this section, we'll first look at the `ListView`. And then, because it is such a versatile control, we'll spend the remainder of the section diving into the `GridView`, including different ways of presenting the items, and working with groups.

10.2.1 Vertical lists

Many apps show data in a single column. Consider the Mail app. In that app, messages are displayed in a list on the left. Similarly, Twitter clients like MetroTwit show several different columns of Tweets. In those cases, the column of data is displayed in a `ListView`.

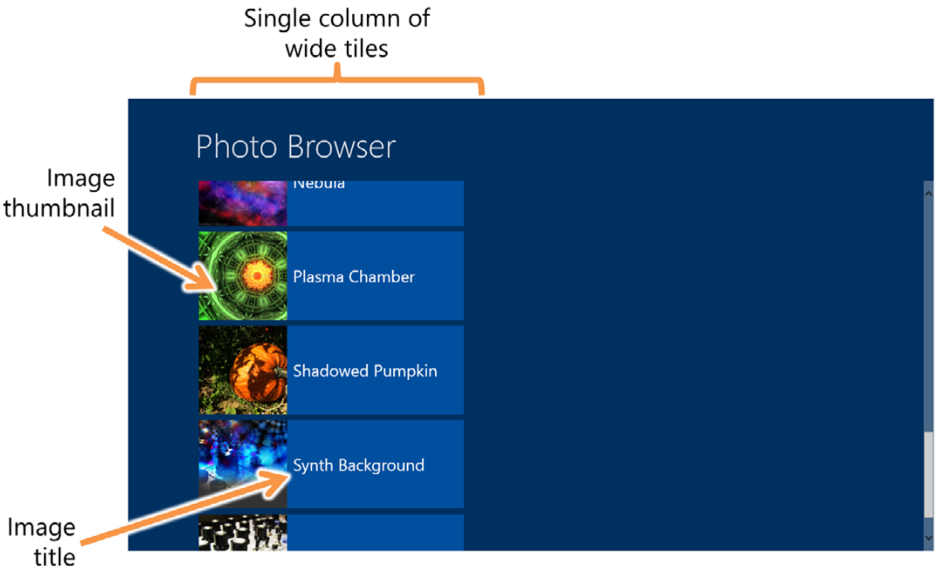

Single column of wide tiles
Image thumbnail
Image title
Photo Browser
Nebula
Plasma Chamber
Shadowed Pumpkin
Synth Background

Figure 10.5 A single `ListView` showing a column of items on the page

Figure 10.5 shows a `ListView` used in this chapter's app to display photos and their names. Note the scrollbar on the right, as well as the way the images are formatted in a single column.

To create the layout shown in figure 10.5, use the XAML from the listing that follows. Paste this into the MainPage.xaml file in the placeholder spot. Leave all the surrounding XAML as is.

Listing 10.6 XAML for the pictures list

```
<ListView ItemsSource="{Binding Source={StaticResource PhotoSource}}"
                        Padding="116,0,40,0"
                        SelectionMode="None">

  <ListView.ItemTemplate>
    <DataTemplate>
      <Grid Width="450" Height="150">
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="150" />
          <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Image Grid.Column="0"
               Stretch="UniformToFill">
          <Image.Source>
            <BitmapImage UriSource="{Binding ImageUri}"
                         DecodePixelHeight="150"/>
          </Image.Source>
        </Image>
```

Bound to CollectionViewSource

Data template

Image with BitmapImage source

```
        <Rectangle Grid.Column="1"
                   Fill="#FF0055AA"
                   Opacity="0.85"/>

        <TextBlock Grid.Column="1"
                   Foreground="White"
                   Margin="10"
                   VerticalAlignment="Center"
                   TextAlignment="Left"
                   TextWrapping="Wrap"
                   FontSize="28"
                   Text="{Binding DisplayName}" />      <-- Display name
      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

There are a number of interesting things in this bit of markup. First is the `ListView` itself. You can see that it's bound to the `CollectionViewSource` resource. Notice that the binding statement includes an embedded `StaticResource` statement. This is how you set up a binding relationship with a data source that's expressed in XAML.

Next is the `DataTemplate` statement inside the `ItemTemplate` property. The data template is what's used to format each list item for display; its `DataContext` is automatically set to the item being displayed. In this example, you use an image, a colored rectangle at 85% opacity, and some white text. The text is bound to the `DisplayName` property.

If you look at the `Image` element in the data template, you'll see that you're not using the `Source` property with the URI directly. Instead, you break it out and have a `BitmapImage` as the source. Why do you do this?

There are two reasons for using a `BitmapImage` in the `Source` property:

- If you want to bind to a URI type, you can't use the `Image` element's `Source` property directly. Instead, you need to use a value converter, a string, or embed an `ImageSource`, which can work with a URI type.
- You want to control the amount of memory used to decode and cache the images. It's a best practice to set the `DecodePixelHeight`, `DecodePixelWidth`, or both when decoding images. (If you set just one of them, the aspect ratio is preserved.) When working with large images, this results in far less memory being used, and it makes it possible to generate thumbnails on the fly without creating completely separate image files.

This shows how, even without resorting to code, you have quite a bit of control over the details of how images are loaded and displayed in XAML.

By default, the `ListView` uses a type of `StackPanel` for its panel. This can be changed by using the `ItemsPanel` property. But before you consider using something like a `WrapGrid`, understand that the `ListView` doesn't work as well as a `GridView` when it comes to items that aren't in a vertical list. In some cases, you'll need to wrestle with scrollbars; in others, the virtualization may not be as efficient. The `ListView` is excellent for vertical lists, but don't attempt to bend it to do something else.

Before you use a `ListView` in your app, consider the interaction approach. If you expect the user to flick through the items quickly, as they might when scanning for a particular entry in a photo-browsing app, a vertical list is not usually appropriate. Why? There are two main reasons: Vertical flicking can trigger the app bar display (next chapter), and most device displays are better suited to horizontal scrolling. Unlike a phone, they're usually held in the hand in landscape orientation, and compared to vertical real estate, they have more horizontal space to allow a greater range of movement without triggering the edges. For horizontal scrolling, the `GridView` is king. Let's look at that next.

10.2.2 Horizontal lists and grids

The view control that really shines in Windows 8 is the `GridView`. The `GridView` enables you to show as many rows and columns of items as the display will support. It automatically handles wrapping (vertically or horizontally, as set through the panel properties) and is built with horizontal scrolling in mind.

Like the `ListView`, the `GridView` has excellent built-in system-level animation. When you swipe or flick, you'll see smooth scrolling as well as an appropriate bounce when you hit the end of the list. You get all of this for free, just for using an OS control.

Figure 10.6 shows the `GridView` of photos for this app.

The `GridView` automatically positions the elements to take up the space available on the screen. If you were to run the app on a higher-resolution display (in the Simulator or on a desktop, for instance), you would see more rows and columns of items.

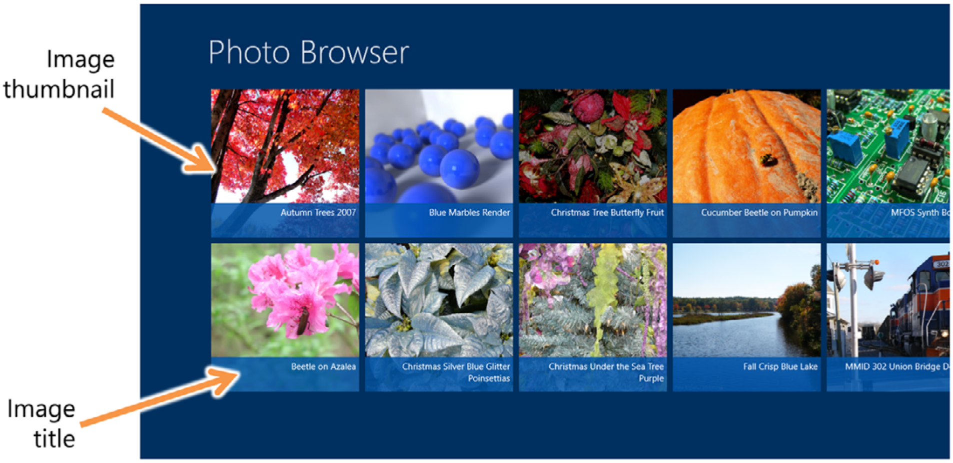


Image thumbnail
Image title
Photo Browser
Autumn Trees 2007
Blue Marbles Render
Christmas Tree Butterfly Fruit
Cucumber Beetle on Pumpkin
MFOS Synth Bo
Beetle on Azalea
Christmas Silver Blue Glitter Poinsettias
Christmas Under the Sea Tree Purple
Fall Crisp Blue Lake
MMID 302 Union Bridge D

Figure 10.6 The Photo Browser app using a `GridView` for display

SQUARE ITEMS OVER MULTIPLE ROWS

The following listing contains the XAML required to implement the uniformly sized grid layout shown in figure 10.6. Replace the previous `ListView` XAML with the contents of this listing.

Listing 10.7 XAML for the grid of square picture tiles

```
<GridView ItemsSource="{Binding Source={StaticResource PhotoSource}}"
          Padding="116,0,40,0">
  <GridView.ItemTemplate>
    <DataTemplate>
      <Grid Width="250" Height="250">
        <Grid.RowDefinitions>
          <RowDefinition Height="*" />
          <RowDefinition Height="0.30*" />
        </Grid.RowDefinitions>

        <Image Grid.Row="0" Grid.RowSpan="2"
               Stretch="UniformToFill">
          <Image.Source>
            <BitmapImage UriSource="{Binding ImageUri}"
                         DecodePixelHeight="250"/>
          </Image.Source>
        </Image>

        <Rectangle Grid.Row="1"
                   Fill="#FF0055AA"
                   Opacity="0.85"/>

        <TextBlock Grid.Row="1"
                   Foreground="White"
                   Margin="5"
                   TextAlignment="Right"
                   TextWrapping="Wrap"
                   FontSize="15"
                   Text="{Binding DisplayName}" />
      </Grid>
    </DataTemplate>
  </GridView.ItemTemplate>
</GridView>
```

Bound to CollectionViewSource

Data template

BitmapImage source again

Semitransparent overlay

Textblock showing DisplayName

The markup here looks remarkably similar to the previous `ListView` example. The API for the controls is almost identical, so it's easy to swap one for the other. As before, this control is bound to the `CollectionViewSource` in the page's resources section. Also as before, all the interesting stuff is going on in the `ItemTemplate` property.

The `DataTemplate` in the `ItemTemplate` property of this control sets up a 250 px x 250 px square and fills it with the image (a more common size is 180 px, but I liked how 250 px looked here). Because the image size has changed from the previous example, you also change the `DecodePixelHeight` to 250 px. Over the image, you have a semitransparent blue rectangle that serves as the backdrop for the `DisplayName` property. The `DisplayName` property is shown in a regular `TextBlock` element with right-aligned text.

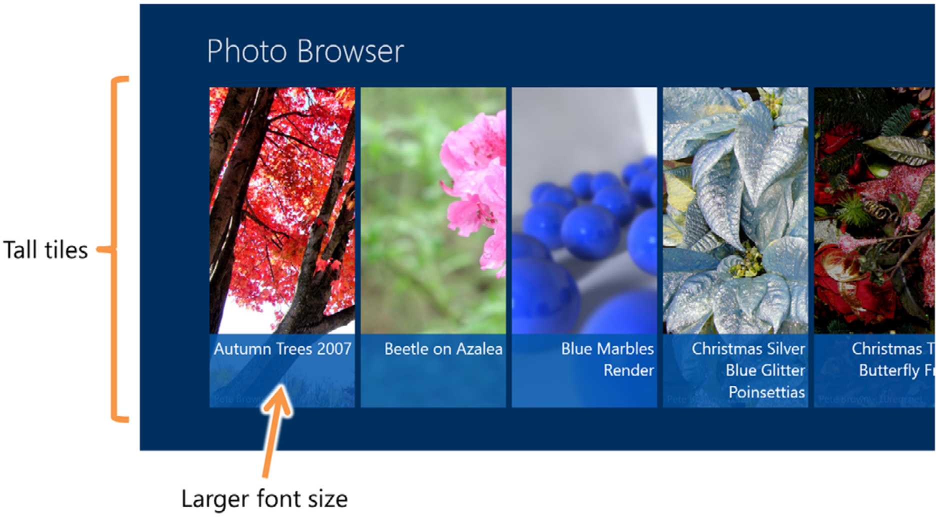


Photo Browser
Autumn Trees 2007
Beetle on Azalea
Blue Marbles Render
Christmas Silver Blue Glitter Poinsettias
Christmas T
Butterfly F
Tall tiles
Larger font size

Figure 10.7 Tall rectangle tiles in the `GridView` control

Everything is then arranged in a grid layout, with uniformly sized items.

> **TIP** Although not shown here, the `GridView` also supports variable-sized elements using the `VariableSizedWrappedGrid`, similar to what you see in the Start page in Windows 8 or the main page of the Windows Store. To support that in your app, you'll need to write code to set up data template selectors, which will set different templates based on what they're binding to; for example, highlighted (large) items and regular (small) items.

RECTANGLE ITEMS WITH A SINGLE ROW

Another common way to lay out items is to use a rectangular layout and a single row. This is a nice departure from the sometimes bland look you get with regular square items. Figure 10.7 shows the same data and the same control, this time with a data template that uses taller tiles.

The next listing shows how to re-create what's shown in figure 10.7. In addition to turning off selection via `SelectionMode="None"`, you change the data template, specifically the grid's height and the decode height of the image.

Listing 10.8 XAML for the grid of tall picture tiles

```
<GridView ItemsSource="{Binding Source={StaticResource PhotoSource}}"
          Padding="116,0,40,0"
          SelectionMode="None">
  <GridView.ItemTemplate>
    <DataTemplate>
      <Grid Width="250" Height="550">
        <Grid.RowDefinitions>
          <RowDefinition Height="*" />
          <RowDefinition Height="0.30*" />
        </Grid.RowDefinitions>
```

New grid height

```
        <Image Grid.Row="0" Grid.RowSpan="2"
               Stretch="UniformToFill">
          <Image.Source>
            <BitmapImage UriSource="{Binding ImageUri}"
                         DecodePixelHeight="550"/>
          </Image.Source>
        </Image>

        <Rectangle Grid.Row="1"
                   Fill="#FF0055AA"
                   Opacity="0.85"/>

        <TextBlock Grid.Row="1"
                   Foreground="White"
                   Margin="5"
                   TextAlignment="Right"
                   TextWrapping="Wrap"
                   FontSize="28"
                   Text="{Binding DisplayName}" />
      </Grid>
    </DataTemplate>
  </GridView.ItemTemplate>
</GridView>
```

New decode height

If you look closely, you'll see that you're relying on the screen resolution to decide how many rows you show. On a high-resolution display, you'll get more rows than shown here. That's okay, and a good use of screen space, but not quite the effect you want. This shows the importance of testing on multiple resolutions. Near the end of this chapter, when I show you Semantic Zoom, you'll see how to swap in a different panel to make sure there is only one row of data.

Go ahead and return the code to the square tiles layout. You'll use that throughout the remainder of this chapter, because it's especially well suited to things like grouping.

10.3 Grouping with the GridView

By far, one of the most interesting features of the `GridView` is its built-in support for grouping. If you look at the start page in Windows, you'll see that tiles are grouped. Similarly, if you look in the Windows Store main screen, you'll see that tiles are again grouped but also have headers. If you click one of those headers, it brings you to the full list of items for that category.

Grouping of items provides natural advantages for making the content more scannable: As humans (no angry letters from cats or dolphins reading my book, please), we like to scan at a high level and then drill down into the details once we find the high-level group we're looking for. It also provides another UI point where you can provide a dedicated UI for looking at a group of items.

In this app, you'll start by implementing basic grouping with straight-text headers and then move on to grouping with active headers that lead to another page. This section will cover the basic grouping; the next section, on the `FlipView`, will include the navigation code.

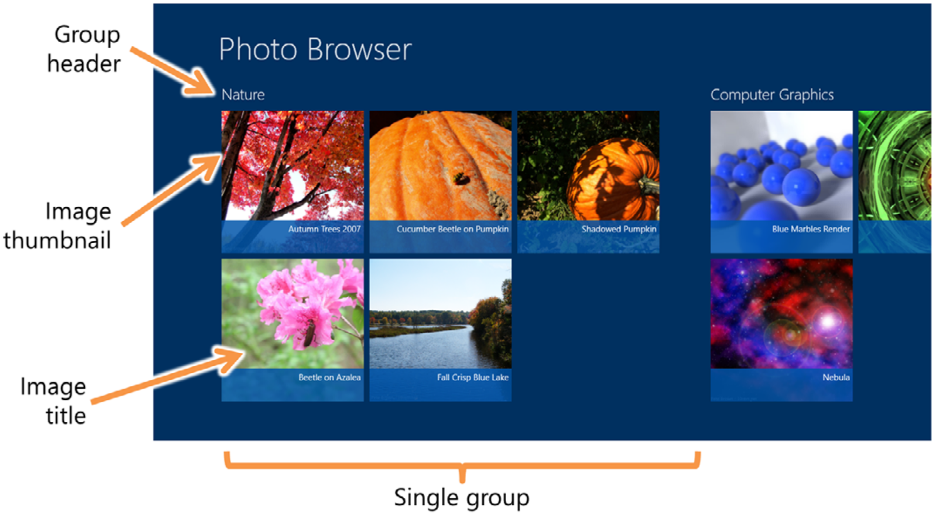

Group header
Image thumbnail
Image title
Photo Browser
Nature
Computer Graphics
Autumn Trees 2007
Cucumber Beetle on Pumpkin
Shadowed Pumpkin
Blue Marbles Render
Beetle on Azalea
Fall Crisp Blue Lake
Nebula
Single group

Figure 10.8 Photos grouped by category, all displayed in a `GridView`

10.3.1 *Grouping in the model and viewmodel*

The data in this app is a bunch of photos and pictures. The model conveniently includes a category for each photo, giving us something to group on. Another possible item to group on would have been the first letter of the filename. What you group on is entirely up to you and the use cases for your app.

Figure 10.8 shows the photos now grouped by category.

Before you can update the UI to support grouping, there's a little back-end work to do. Yes, the photos themselves each have a category, but for the `GridView` to group the items, you need to have a higher-level class that has the group caption information as well as a list of items in that group. The next listing includes the new `PhotoCategory` class. Create this in your model folder.

Listing 10.9 The `PhotoCategory` model class

```
using System.Collections.Generic;

namespace PhotoBrowser.Model
{
  public class PhotoCategory
  {
    public string Category { get; set; }
    public IList<Photo> Photos { get; set; }
  }
}
```

Category title

List of items in that category

Given that this class was created specifically to support something in the UI, equally good arguments could be made to make that a viewmodel and not a model class. See? I'm making sure you have plenty to talk about with your architects.

One interesting aspect of having a separate class like this is you can put just a subset of items into the child collection. This is, for example, how the Windows Store operates: Highlighted items show up in the category, but you need to navigate to the category itself to see everything. This is also an important approach to consider when the items will be virtualized, because virtualization and grouping make for a difficult pairing.

Once you have the model object created, you need a way to populate it. The following listing shows the updates to the `MainViewModel` class. Replace the existing `Photos` collection (and backing property) and `LoadPhotos` method with this code.

Listing 10.10 `ViewModel` updates for grouping

```
private ObservableCollection<PhotoCategory> _photos =
    new ObservableCollection<PhotoCategory>();

public ObservableCollection<PhotoCategory> Photos
{
    get { return _photos; }
}

public void LoadPhotos()
{
    var photos = ImageService.Current.GetPhotos();

    var groups = photos.GroupBy(p => p.Category)
        .Select(p => new PhotoCategory()
            {
                Category = p.Key,
                Photos = p.ToList()
            });

    Photos.Clear();

    foreach (var g in groups)
        Photos.Add(g);
}
```

Collection of PhotoCategory

LINQ grouping by category

Collection loading

Grouping is performed with a little LINQ code in the viewmodel. The query groups by photo category, and for each category it creates a `PhotoCategory` instance with the group key (the `Category` property of the `Photo`) as the `Category` property of the `PhotoCategory` and the list of items in the group as the child collection for the `PhotoCategory`. Now you have something that represents the category itself and something else that gives you everything you need to display for that category.

The viewmodel now exposes a collection of `PhotoCategory` instances rather than a collection of `Photo` instances. The `GridView` will bind to these categories rather than to the photos themselves. But, to make that happen, you first need to update the `CollectionViewSource`.

10.3.2 *Grouping at the UI layer*

The UI needs to be updated to support grouping. This is one reason I went with a `CollectionViewSource` in this chapter rather than binding the `GridView` directly to the `Photos` property in the model.

The following is the updated `CollectionViewSource` with the key grouping properties.

Listing 10.11 Grouped photos `CollectionViewSource` XAML

```
<Page.Resources>
  <CollectionViewSource x:Key="PhotoSource"
                        IsSourceGrouped="True"
                        ItemsPath="Photos"
                        Source="{Binding Photos}">
  </CollectionViewSource>
</Page.Resources>
```

Yes, the data is grouped

Property name for child collection

There are two changes to the `CollectionViewSource`. First is the `IsSourceGrouped` property. If this property is set to true, the `CollectionViewSource` knows that there will be a child collection of items. It looks for this child collection by inspecting the `ItemsPath` property. Set this property to the name of your child collection property.

If you run the app right now, the grouping still won't be correctly represented. That's because you need to provide templates to handle the display of the groups. The next listing shows the updated `GridView` with the new header and group properties in place.

Listing 10.12 Grouped photos `GridView` XAML

```
<GridView ItemsSource="{Binding Source={StaticResource PhotoSource}}"
          Padding="116,0,40,0"
          SelectionMode="None">

  <GridView.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel Orientation="Horizontal"/>
    </ItemsPanelTemplate>
  </GridView.ItemsPanel>

  <GridView.GroupStyle>
    <GroupStyle>
      <GroupStyle.HeaderTemplate>
        <DataTemplate>
          <Grid Margin="5,0,0,15">
            <TextBlock Text="{Binding Category}"
                       Style="{StaticResource GroupHeaderTextStyle}" />
          </Grid>
        </DataTemplate>
      </GroupStyle.HeaderTemplate>

      <GroupStyle.Panel>
        <ItemsPanelTemplate>
          <VariableSizedWrapGrid Orientation="Vertical"
                                 Margin="0,0,80,0"/>
        </ItemsPanelTemplate>
      </GroupStyle.Panel>

    </GroupStyle>
  </GridView.GroupStyle>
```

Panel to hold groups

Header template

Panel to hold photos

```
  <GridView.ItemTemplate>
    <DataTemplate>
      <Grid Width="250" Height="250">
        <Grid.RowDefinitions>
          <RowDefinition Height="*" />
          <RowDefinition Height="0.30*" />
        </Grid.RowDefinitions>

        <Image Grid.Row="0" Grid.RowSpan="2"
               Stretch="UniformToFill">
          <Image.Source>
            <BitmapImage UriSource="{Binding ImageUri}"
                         DecodePixelHeight="250"/>
          </Image.Source>
        </Image>

        <Rectangle Grid.Row="1"
                   Fill="#FF0055AA"
                   Opacity="0.85"/>

        <TextBlock Grid.Row="1"
                   Foreground="White"
                   Margin="5"
                   TextAlignment="Right"
                   TextWrapping="Wrap"
                   FontSize="15"
                   Text="{Binding DisplayName}" />
      </Grid>
    </DataTemplate>
  </GridView.ItemTemplate>
</GridView>
```

ItemTemplate as before

One important thing to notice in this listing is that from the `GridView` control's perspective, the items are now the groups themselves. That's why the `ItemsPanel` property, which normally would change the panel used to lay out individual items, is used to lay out the groups. For convenience, the `ItemTemplate` is still the template that's used to lay out the individual photos (in fact, it's identical to the one you've used previously). This mismatch between panels and templates can be a little confusing at first; take a look at table 10.1 to get it all straight.

Table 10.1 The important template properties of the `GridView` used when grouping items

Template	Role when grouping
`ItemsPanel`	Panel that lays out all groups
`GroupStyle.HeaderTemplate`	Template for the header to display above the group contents
`GroupStyle.Panel`	Panel that lays out the individual items in the group
`ItemTemplate`	The template for the individual items in the group

The `GridView` is my favorite list control in Windows 8. It has tons of flexibility, the animations are smooth and pleasing, and being able to group the data makes for a great display and great usability. I like that everything is in the template, so you can make the control look any way you want. If you want tall elements, you can do that. If you want squat rectangles or uniform squares, you can do that.

The examples I've shown so far have only a single page. It's time to liven up the app just a little and add a second page. This is also a great way to show off the `FlipView` control and the basics of the navigation API combined with clickable group headers.

10.4 FlipView and navigation

Each of the view controls is designed to display a list of items. The `ListView` displays them vertically; the `GridView` displays them (generally) in a horizontal grid layout. The final view control, the `FlipView`, displays them one at a time, with the ability to navigate forward and backward through the list.

The `FlipView` works equally well with mouse or touch. On a touch screen, you swipe left and right to scroll through the list. On a mouse-based system, you use the Previous and Next buttons to do the same.

In this section, you'll use the `FlipView` to display the photos within a single group. Figure 10.9 shows what the interface will look like.

Throughout the chapters, you've seen a function called `OnNavigatedTo`. I haven't done much with it so far. This function is part of the navigation support built into WinRT. As the name suggests, it fires off whenever a page has been navigated to. In this chapter, we'll also look at the API call used to initiate the navigation to a page and the code required to navigate back to the previous page using a button at the top-left

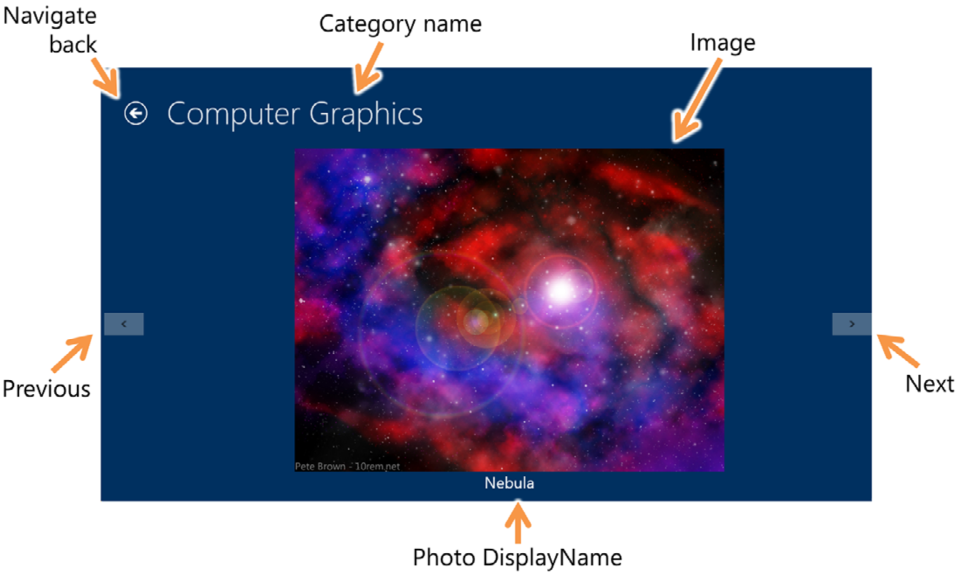


Figure 10.9 The `FlipView` control on the new page. I used a mouse when taking this image, so the previous and next buttons are visible.

corner of the screen. In addition to the markup and code-behind required to support this, you'll need to add a new page and a new viewmodel.

10.4.1 *Viewmodel*

I generally subscribe to the approach of one-to-one relationships between viewmodels and pages. I also think each page should have a viewmodel, rather than follow a different pattern for even simple pages.

The viewmodel for this page exists only to expose the `PhotoCategory` instance the items on the page will be bound to. In the ViewModel folder, create a new class named `CategoryBrowserViewModel` and paste into it the code that follows.

Listing 10.13 New category browser page viewmodel

```
using GalaSoft.MvvmLight;
using PhotoBrowser.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PhotoBrowser.ViewModel
{
  public class CategoryBrowserViewModel : ViewModelBase
  {
    private PhotoCategory _category;
    public PhotoCategory Category
    {
      get { return _category; }
      set
      {
        Set<PhotoCategory>(() => Category, ref _category, value);
      }
    }
  }
}
```

PhotoCategory instance

The viewmodel for the category browser page has a single property: `Category`. The `Category` will be bound to from the controls in the UI (no `CollectionViewSource` this time) and will be assigned to by the code in the `OnNavigatedTo` event.

10.4.2 *Category browser page*

The category browser page follows the same general pattern we've had so far. The top has a title and the body is a view control. Unlike the previous examples, the page title will be pulled from a value in the viewmodel. There's also now a Back button, and the view control will be bound directly to a property of the viewmodel, not to a `CollectionViewSource`.

In the root of the project, add a new blank page named `CategoryBrowserPage` and paste into it the XAML from the next listing.

Listing 10.14 Category browser page XAML

```
<Page
  x:Name="pageRoot"
  x:Class="PhotoBrowser.CategoryBrowserPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PhotoBrowser"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid Background="#FF003060">
    <Grid.RowDefinitions>
      <RowDefinition Height="140"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Grid Grid.Row="0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="116"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
```

Back button for navigation

```
      <Button x:Name="backButton"
         IsEnabled="{Binding Frame.CanGoBack, ElementName=pageRoot}"
              Click="OnBackButtonClick"
              Style="{StaticResource BackButtonStyle}"/>
```

Page title bound to category name

```
      <TextBlock x:Name="pageTitle"
                 Text="{Binding Category.Category}"
                 Grid.Column="1" IsHitTestVisible="false"
                 Style="{StaticResource PageHeaderTextStyle}"/>
    </Grid>
```

FlipView bound to Photos

```
    <Grid Grid.Row="1">
      <FlipView ItemsSource="{Binding Category.Photos}"
                Padding="116,0,40,0">
```

ItemTemplate

```
        <FlipView.ItemTemplate>
          <DataTemplate>
            <Grid>
              <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="50" />
              </Grid.RowDefinitions>

              <Image Grid.Row="0"
                     Stretch="Uniform">
                <Image.Source>
```

BitmapImage image source

```
                  <BitmapImage UriSource="{Binding ImageUri}" />
                </Image.Source>
              </Image>

              <TextBlock Grid.Row="1"
                         Foreground="White"
                         Margin="5"
```

```
                            TextAlignment="Center"
                            TextWrapping="Wrap"
                            FontSize="28"
                            Text="{Binding DisplayName}" />
             </Grid>
           </DataTemplate>
         </FlipView.ItemTemplate>
       </FlipView>
     </Grid>
   </Grid>
</Page>
```

Unlike the main page in the app, this page sports a Back button. When using the default templates, many developers leave the Back button on the main page of the app. It really shouldn't be there unless you can do something with it. For that reason, I didn't put one on the main page. But this secondary page can certainly use a way to navigate back to the main page.

Each page in the app is loaded into a frame. You can think of the frame as the host for pages. The `Frame` property of the page is a pointer back to that hosting frame. The Back button is enabled or disabled based on the `CanGoBack` property of that frame.

> **TIP** To see how the frame gets loaded, look at the `OnLaunched` method in App.xaml.cs. You'll see the frame get created and the first page get loaded into it.

Unlike the `GridView` used on the main page, the `FlipView` control on this page doesn't need to know anything about grouping or other advanced features. For that reason, it can be bound directly to properties on the viewmodel—the data context for the page.

But how does the viewmodel get populated? For that, you rely on a little code-behind code related to the navigation API. The `OnNavigatedTo` event and the rest of the CategoryBrowserPage.xaml.cs content are shown here.

Listing 10.15 Category browser page code-behind

```
using PhotoBrowser.Model;
using PhotoBrowser.ViewModel;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace PhotoBrowser
{
  public sealed partial class CategoryBrowserPage : Page
  {
    private CategoryBrowserViewModel _vm =
      new CategoryBrowserViewModel();

    public CategoryBrowserPage()
    {
      this.InitializeComponent();

      DataContext = _vm;
    }
```

Create new viewmodel

Set page data context

```
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
      var category = e.Parameter as PhotoCategory;

      _vm.Category = category;
    }

    private void OnBackButtonClick(object sender,
                 Windows.UI.Xaml.RoutedEventArgs e)
    {
      Frame.GoBack();
    }
  }
}
```

Get navigation parameter

Pass category to viewmodel

The viewmodel and data context setup code looks like what you've already used with the main page. Beyond that, there are two methods worth digging into.

The first is the `OnNavigatedTo` function. This event is fired off when the page is navigated to. You should do any nonstatic page setup in this event because page instances may be reused and navigated to multiple times. So, if the data for each instance of the page is different, you should ensure it's set in the `OnNavigatedTo` override. In this case, you use the `OnNavigatedTo` event to populate the viewmodel with the category, which is passed in from the main page.

The second is the `OnBackButtonClick` event handler. This is wired to the Back button on the page. When you click the button, the code calls the `GoBack` method of the `Frame`, navigating the whole frame back to the main page of the app.

10.4.3 Updated MainPage

The `MainPage` XAML also needs to be updated to provide navigation to the browser page. You'll do this by making it so when you click the header of a category, you can go to the new page to view all the images in that category.

The following listing has the updated header template to be pasted into the MainView.xaml markup for the `GridView`. Replace the old header template with this version.

Listing 10.16 Updated MainPage.xaml `GridView` header template

```
<GroupStyle.HeaderTemplate>
  <DataTemplate>
    <Grid Margin="5,0,0,15">
      <Button Click="OnCategoryHeaderClick"
              Style="{StaticResource TextPrimaryButtonStyle}">
        <TextBlock Text="{Binding Category}"
                   Style="{StaticResource GroupHeaderTextStyle}" />
      </Button>
    </Grid>
  </DataTemplate>
</GroupStyle.HeaderTemplate>
```

Navigation button

Now, instead of just a `TextBlock` displaying the category name, you have a button that calls an event handler in the code-behind to navigate to the new page. The next listing includes the event handler code.

Listing 10.17 Updated MainPage.xaml.cs code-behind for navigation

```
private void OnCategoryHeaderClick(object sender, RoutedEventArgs e)
{
  var category = (sender as FrameworkElement).DataContext;

  this.Frame.Navigate(
      typeof(CategoryBrowserPage),
      category);
}
```

Get selected category

Navigate to page, passing category

When the category header is clicked, you first get the `DataContext` of the header. Because of the way data templates work, the context is the `PhotoCategory` instance you're interested in.

You then take that instance and navigate to the `CategoryBrowserPage`, passing in the `PhotoCategory` instance as a parameter. Notice how the Navigate API works: You pass in the type representing the page to load and optionally pass in a parameter of any type you want. In this way, you can pass a viewmodel instance or something as simple as the `PhotoCategory` model object used here.

Run the app now. You should be able to click a category header and see all the items in the category. Once you're on the category browser page, use your mouse or touch to scroll through the items, left to right. When you've finished browsing, hit the Back button to return to the main page.

So now you have grouped lists of items, a scrollable list of items, and navigation between them. For this chapter, there's one last thing you need to add to the app, and that's Semantic Zoom.

10.5 Semantic Zoom

Stop what you're doing (well, after you finish reading this paragraph), go to the desktop on your computer, and hit the Windows key and then the plus key. The screen will zoom in. This is optical zoom: Everything gets bigger, and each pixel from the zoomed-in view can be traced back to the original view. To get back, hit the Windows key and then the minus key.

Semantic Zoom differs from optical zoom in that the zoomed-in view is only logically related to the zoomed-out view. It shows a higher-level view of the data in order to make it easier to navigate. To see this in action, pinch the screen on the Start page of Windows. You'll see a higher-level view of the app tiles (if you use a mouse, there's also a zoom-out button at the bottom right). Figure 10.10 shows the zoomed-in and zoomed-out views for this chapter.

The zoomed-out view for this app is implemented on MainPage.xaml as a single-row `GridView` with a horizontal `StackPanel` of image thumbnails and category titles. To make Semantic Zoom work, both the zoomed-out and zoomed-in views must be

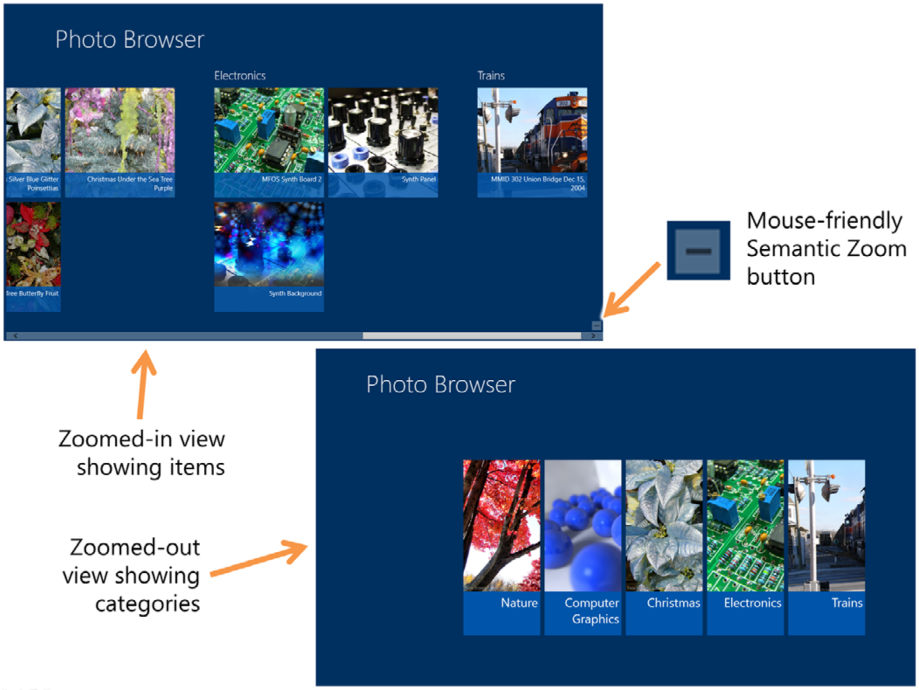


Photo Browser
Electronics
Trains
Silver Blue Glitter Poinsettias
Christmas Under the Sea Tree Purple
MFOS Synth Board 2
Synth Panel
MMID 302 Union Bridge Dec 15, 2004
Tree Butterfly Fruit
Synth Background
Mouse-friendly Semantic Zoom button
Zoomed-in view showing items
Zoomed-out view showing categories
Photo Browser
Nature
Computer Graphics
Christmas
Electronics
Trains

Figure 10.10 Semantic Zoom in action. The zoomed-in view shows the details for each category. The zoomed-out view shows just the category name and a representative image. The mouse-friendly button is enabled through a property on the `SemanticZoom` control.

enclosed in a single `SemanticZoom` control as shown here. This code goes in MainPage.xaml around the current `GridView`. (Don't overwrite the `GridView`; paste it inside the `ZoomedInView` template.)

Listing 10.18 Semantic Zoom control XAML

```
<Grid Grid.Row="1">
  <SemanticZoom IsZoomOutButtonEnabled="True">
    <SemanticZoom.ZoomedInView>

    </SemanticZoom.ZoomedInView>

    <SemanticZoom.ZoomedOutView>

    </SemanticZoom.ZoomedOutView>
  </SemanticZoom>
</Grid>
```

Paste existing GridView here

New GridView will go here

The `IsZoomOutButtonEnabled` property controls whether or not the little minus button shows up on the bottom right of the control.

The zoomed-out view is a `GridView` containing a high-level look at the data. It follows all the same patterns you've been doing with the `GridView` throughout the chapter with a few notable additions. The following listing includes the `GridView` markup to be pasted into the zoomed-out view section of the `SemanticZoom` control.

Listing 10.19 Zoomed-out view XAML

```
<GridView x:Name="SemanticZoomedOutView"
          ScrollViewer.IsHorizontalScrollChainingEnabled="False"
          Padding="116,0,40,0"
          SelectionMode="None">
  <GridView.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Center"
                  Margin="0,0,10,0"/>
    </ItemsPanelTemplate>
  </GridView.ItemsPanel>

  <GridView.ItemTemplate>
    <DataTemplate>
      <Grid Width="175">
        <Grid.RowDefinitions>
          <RowDefinition Height="*" />
          <RowDefinition Height="300" />
          <RowDefinition Height="100" />
          <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Image Grid.Row="1"
               Stretch="UniformToFill">
          <Image.Source>
            <BitmapImage UriSource="{Binding Group.Photos[0].ImageUri}"
                         DecodePixelHeight="300"/>
          </Image.Source>
        </Image>

        <Rectangle Grid.Row="2"
                   Fill="#FF0055AA"
                   Opacity="1.0"/>

        <TextBlock Grid.Row="2"
                   Foreground="White"
                   Margin="5"
                   TextAlignment="Right"
                   TextWrapping="Wrap"
                   FontSize="28"
                   Text="{Binding Group.Category}" />
      </Grid>
    </DataTemplate>
  </GridView.ItemTemplate>
</GridView>
```

Required for Semantic Zoom

Enforce single row

ItemTemplate

First image in category

Category name

The `GridView` has a `ScrollViewer` instance in its template. This is what provides horizontal scrolling capabilities. In order to have Semantic Zoom work correctly, the `IsHorizontalScrollChainingEnabled` attached property of the `ScrollViewer` must be set to false. This property controls whether or not the `ScrollViewer` chains the scrolling from the parent to the child. In the case of Semantic Zoom, the zoomed-out view is a different size and may have different proportions for the items between the zoomed-out and zoomed-in views, so you don't want the chaining to be in place.

To show the first image from the category, the `Image` element's source is set to the first photo in the `Photos` collection of the `PhotoCategory` instance. `Group` is the instance property name on the `CollectionViewSource`. In our case, it's the `PhotoCategory` object.

Note also that the `GridView` doesn't have any `ItemsSource` property set. That's because for Semantic Zoom to work, the zoomed-out view must be bound to the same `CollectionViewSource` as the zoomed-in view, but that doesn't work well in markup. So, instead, you set that in the code-behind, as shown in the following listing.

Listing 10.20 MainPage.xaml.cs code-behind to set zoomed-out view's `ItemsSource`

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  _vm.LoadPhotos();

  var cvs = Resources["PhotoSource"] as CollectionViewSource;

  SemanticZoomedOutView.ItemsSource =
        cvs.View.CollectionGroups;
}
```

Get CollectionViewSource resource

Link both GridView data sources

You could have avoided all of this and simply bound the zoomed-out view to a property on the viewmodel. The problem is Semantic Zoom requires that the two views be related by being bound to the same data source. Sure, it will work without that, but you'll notice that selecting an item in the zoomed-out view won't have any effect on the positioning of the elements in the zoomed-in view. When they are connected, selecting an item in the zoomed-out view brings that group into view on the zoomed-in view.

Semantic Zoom is a powerful pattern for navigating large amounts of data. You wouldn't want your users scrolling for 10 minutes just to find a piece of data, so instead you logically group the items and expose that grouping as the zoomed-out high-level view.

10.6 Summary

The view controls are some of the most interesting and most useful controls in Windows 8. Unless you're building something with a completely unique mode of interaction (like a game), you'll probably end up using one or more of these in your own apps. The demo app in this chapter was designed specifically to show off these controls, but at a high level, it resembles any number of data-consumption apps you may find in the Windows Store.

When you want to show items in a vertical list, the `ListView` is the control to use. You can sometimes coerce it to show items in multiple columns or in a horizontal layout, but that's really not the best use of this control. For those, you'll want to use a `GridView`.

The `GridView` can do everything the `ListView` can do (although its virtualization optimizations differ), as well as show items in regular grids, irregular grids, or groups. The grouping capability is especially interesting, because it makes it possible to provide structure to the visualization above and beyond the flat grid.

A common pattern with grouped `GridView` data is to use the header to navigate to a group details page. For this demo app, the header navigated to a page containing a `FlipView`—the third and final type of view control. The `FlipView` control is optimized for showing a single element at a time, while enabling you to scroll horizontally through the items.

Another exciting pattern that builds on the `GridView` is the Semantic Zoom pattern. Semantic Zoom differs from optical zoom in that the zoomed-in and zoomed-out views may be quite different; they aren't related to each other in an optical or pixel way but rather in a logic data way. WinRT XAML even includes a dedicated `SemanticZoom` control, which encapsulates the logic required for switching between the two logical views. Semantic Zoom is an excellent pattern you'll want to use if you have large amounts of data to sift through in your apps.

In the next chapter, you'll build on the sample covered here to add in other essential Windows 8 UI components such as the app bar, tiles, and more.

11

The app bar

This chapter covers

- The app bar and buttons
- Using the app bar for navigation
- Popups and menus

It may be hard to imagine applications without toolbars, but prior to the mid–Windows 3.1 timeframe, they were uncommon in GUIs. The original Apple Macintosh (and the Xerox technology it was based on) and clearly derivative GEOS, all with their single menu bar at the top, plus the Amiga Workbench and Windows 1.x with flat menus per window together established the menu navigation standard. None of them made real use of toolbars.

Eventually, applications got more complicated, and their menus became overly complex, making it difficult to find common tasks. With the rise in screen resolution, more screen real estate became available, and toolbars became commonplace. Eventually this got out of control, especially in complex applications like Microsoft Word where, if you displayed all the available toolbars, you'd end up with a tiny sliver of real estate available for content. I remember visiting my parents one year and their Word toolbars looked just like that, but they had no idea how they got that way. Oh, and don't get me started on all the installed Internet Explorer toolbars they had. Even in a well-maintained toolbar, the number of options got to be so many that the same problems we had with discovering items in complex menus started to surface in toolbars. We simply pushed the problem around.

Microsoft countered by introducing the Office Ribbon. This was, and continues to be, an effective way to manage toolbar options in that it relies on context-sensitive display of whole tabs as well as individual items on tabs. It also makes use of labels, space permitting, to ensure that icons can be identified. It does all this while using approximately the same amount of vertical space as was typically allocated to menus and two rows of toolbars.

Modern Style apps in Windows generally don't have the complexity of these full desktop apps and therefore don't need the vast array of menu and toolbar options. But the focus of content over chrome (buttons, toolbars, decorations) and the requirement for a touch-friendly UI on small screens offer their own challenges.

For Windows apps, the app bar is the answer to these challenges. It's where you'd put functionality normally thought of as menu and toolbar options in desktop apps. Following the Windows design aesthetic, it uses simple iconography. Learning from the work with the Ribbon, it supports easily readable text labels for buttons. Also taking a page (or, ahem, tab) from the Ribbon and earlier work with context-sensitive toolbars, it's expected to be used in a context-sensitive way, minimizing the amount of clutter. Finally, following the Windows focus on content over chrome, the app bar is

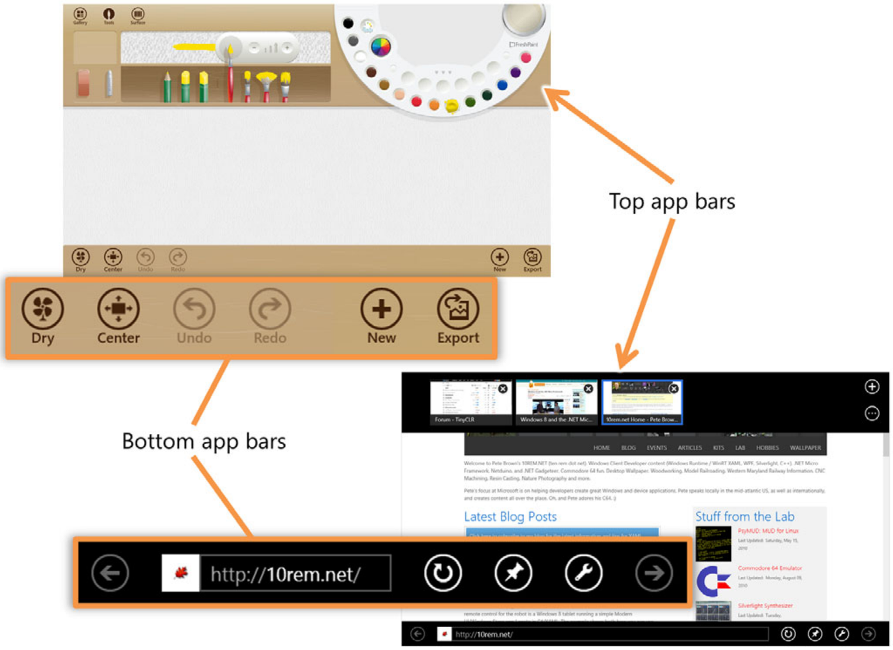


Figure 11.1 Two very different approaches to app bar functionality. Fresh Paint uses the top app bar for the tools. Internet Explorer uses the top app bar for navigation between tabs. Both use the bottom app bar for contextual commands.

typically invisible until activated by a swipe on the screen, a right-click with the mouse, the app key (the one on the right that looks like a typical context menu) or the Windows + Z key combination. Figure 11.1 shows several app bars in use.

Continuing with the PhotoBrowser demonstration app from the previous chapter, in this chapter you'll learn how to use the app bar. First, we'll look at how to create an app bar and add app bar-specific buttons that perform item and page tasks plus techniques and properties for controlling when the app bar is visible and whether or not it is "sticky." Next, we'll look at a common use of the top app bar: navigation. Finally, we'll look at expanding on the interactivity of the app bar by adding pop-up UI elements like menus. First, we have a few updates to the PhotoBrowser app to make it work in this chapter.

11.1 Project updates

This chapter continues adding features and functionality to the PhotoBrowser app. The focus in this chapter is, of course, the app bar. If you're using the downloadable source code, pick up the version from the previous chapter and start there.

To support some of the concepts in this chapter, you'll have to do a little refactoring and make changes to the source code. The first change is to update the `CategoryBrowserViewModel` so that it can be passed whole during page navigation. This will enable you to provide from the main page more information than just the single category.

Our first listing has the new version of the viewmodel.

Listing 11.1 Updated `CategoryBrowserViewModel` for navigation

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using PhotoBrowser.Model;
using PhotoBrowser.Services;
using System.Collections.ObjectModel;

namespace PhotoBrowser.ViewModel
{
  public class CategoryBrowserViewModel : ViewModelBase
  {
    private ObservableCollection<PhotoCategory> _allCategories =
      new ObservableCollection<PhotoCategory>();

    public ObservableCollection<PhotoCategory> AllCategories
    {
      get { return _allCategories; }
      set
      {
        Set<ObservableCollection<PhotoCategory>>(
          () => AllCategories, ref _allCategories, value);
      }
    }

    private PhotoCategory _category;
```

All categories

```
    public PhotoCategory Category
    {
      get { return _category; }
      set
      {
        Set<PhotoCategory>(() => Category, ref _category, value);

        //DeleteAllPhotosCommand.RaiseCanExecuteChanged();

        if (Category != null &&
          Category.Photos != null &&
          Category.Photos.Count > 0)
          SelectedPhoto = Category.Photos[0];
      }
    }

    private Photo _selectedPhoto;
    public Photo SelectedPhoto
    {
      get { return _selectedPhoto; }
      set
      {
        Set<Photo>(() => SelectedPhoto, ref _selectedPhoto, value);
        //RotateSelectedPhotoCommand.RaiseCanExecuteChanged();
        //LikeSelectedPhotoCommand.RaiseCanExecuteChanged();
        //DislikeSelectedPhotoCommand.RaiseCanExecuteChanged();
      }
    }

  }
}
```

Current category

You'll uncomment this line later

The currently selected photo

You'll uncomment these later

Inside the `Category` setter is a single commented-out line. When you add that command object later in this chapter, you'll uncomment that line.

The category browser page in this chapter requires access to all the categories from the main page. In order to provide that during navigation, as well as the selected category, you have to pass the entire viewmodel to the page. The code to do this is shown here.

Listing 11.2 MainPage.xaml.cs updates to navigation code

```
private void OnCategoryHeaderClick(object sender, RoutedEventArgs e)
{
  var category =
    (sender as FrameworkElement).DataContext as PhotoCategory;

  var browserViewModel = new CategoryBrowserViewModel();

  browserViewModel.AllCategories = _vm.Photos;
  browserViewModel.Category = category;

  this.Frame.Navigate(
    typeof(CategoryBrowserPage),
    browserViewModel);
}
```

Create viewmodel

Assign categories

Navigate with viewmode

The new navigation code takes ownership of creating the category browser page's viewmodel and providing it to that page during navigation. I've seen this pattern followed in a number of MVVM apps where a master page provides the viewmodel for a child page. This pattern works only if the child page can't be navigated to without first going through the master page.

Note that you'll also need to add a `using` statement for the `Model` namespace or else the reference to `PhotoCategory` won't compile.

Because of these changes, you also need to modify the code-behind in the CategoryBrowserPage.xaml.cs file. The following listing has the new version.

Listing 11.3 CategoryBrowserPage.xaml.cs updates to navigation code

```
using PhotoBrowser.Model;
using PhotoBrowser.ViewModel;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace PhotoBrowser
{
  public sealed partial class CategoryBrowserPage : Page
  {
    private CategoryBrowserViewModel _vm;

    public CategoryBrowserPage()
    {
      this.InitializeComponent();
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
      var vm = e.Parameter as CategoryBrowserViewModel;

      _vm = vm;

      DataContext = _vm;
    }

    private void OnBackButtonClick(object sender,
              Windows.UI.Xaml.RoutedEventArgs e)
    {
      Frame.GoBack();
    }
  }
}
```

Do not create viewmodel instance

Get assigned instance

In this new version of the code, the `CategoryBrowserPage` no longer constructs its own viewmodel. Instead, the viewmodel is assigned from the `OnNavigatedTo` method, provided by the master page. Once the viewmodel is obtained, it's set as the data context for the entire page.

The changes made to these pages and viewmodels now support the additional detail required for you to use the app bar in this chapter. Changing the navigation model to pass in the viewmodel from the main page is something you may find yourself doing in your own apps as well.

Now that everything is in place, you'll build out the bottom app bar and give the UI some much-needed interactivity.

11.2 Controls on the bottom app bar

The app bar control is surfaced through the `TopAppBar` and `BottomAppBar` properties of the `Page`. Because it's simply a `ContentControl` with a few additional properties and behaviors, you may put any type of control in it. For example, I've seen everything from specialized buttons to range-selection graphs to full `ListView` and `GridView` controls filled with items. Figure 11.1 shows two very different approaches to app bar layout, each appropriate to the type of app.

Notice that although both apps use the app bars differently, they are both recognizable as app bars. They both appear from the top and bottom in the same way and dismiss in the same way. The behavior is consistent, as is the general approach to how the buttons are displayed. You can see that you have a lot of creative freedom in how the app bars work.

This section covers most of what you need to know to work with the app bar. First, you'll create a bottom app bar and then add appropriately styled buttons to it. You'll also look at how to add separators and buttons for which there exists no built-in style. Then, using what you learned in chapter 9, you'll wire those buttons to the viewmodel using commands.

The commands will do a few interesting things. The easiest of the commands will increment and decrement a rating, much like "likes" and "dislikes" on something like YouTube. Another command will rotate the image by setting an angle in the model object. As part of that, you'll also implement a render transform on the photo. The final command will delete from memory every photo in the category and then the category itself. You'll implement a new function in the `ImageService` to make that easier.

Finally, we'll wrap up this section with a look at how to make the app bar visible by default and sticky, something you may want to do to have it behave more like the toolbars in desktop applications.

11.2.1 Adding and styling buttons

For many apps, the app bar will simply be a container for buttons, just as the toolbar was. Where you place the buttons, and how you style them, is an important part of creating an effective app bar. When deciding where to put buttons, a good rule of thumb is to have selection-based buttons on the left and page-based buttons on the right. For styling, the StandardStyles.xaml in the Common folder has a number of built-in app bar button styles to get you started.

You'll use three of the built-in button styles and then add a fourth one based on your own glyph. Figure 11.2 shows what your completed app bar will look like.

Listing 11.4 shows the app bar portion of the markup for the app. The XAML goes anywhere in CategoryBrowserPage.xaml in the `Page` element, because `BottomAppBar` is

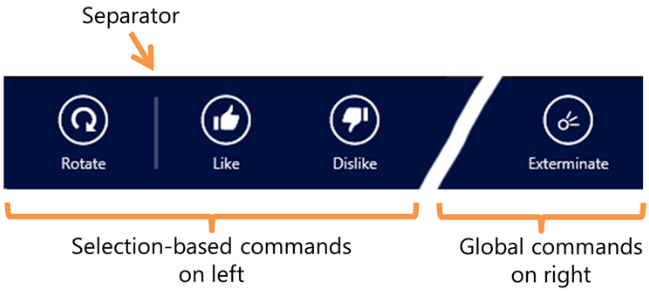

Separator
Rotate
Like
Dislike
Exterminate
Selection-based commands on left
Global commands on right

Figure 11.2 The bottom app bar. Selection-based commands are on the left, while the right side holds commands that operate even when nothing is selected. The separator was created with a semitransparent `Rectangle`; the buttons all use Segoe UI glyphs.

a property of the `Page`, broken out here using property element syntax. I tend to put the definition either at the very top of the file, just under the opening `Page` tag, or at the very bottom of the file, just before the closing `Page` tag.

Listing 11.4 Bottom app bar with styled buttons

```
<Page.BottomAppBar>
  <AppBar Background="#FF001040">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>

      <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Left"
                  Grid.Column="0">
        <Button Style="{StaticResource RotateAppBarButtonStyle}"
                Command="{Binding RotateSelectedPhotoCommand}"/>

        <Rectangle Margin="5,15,5,15"
                   Stroke="White" StrokeThickness="2"
                   VerticalAlignment="Stretch"
                   Width="1"
                   Opacity="0.25"/>

        <Button Style="{StaticResource LikeAppBarButtonStyle}"
                Command="{Binding LikeSelectedPhotoCommand}" />

        <Button Style="{StaticResource DislikeAppBarButtonStyle}"
                Command="{Binding DislikeSelectedPhotoCommand}" />
      </StackPanel>

      <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Right"
                  Grid.Column="1">
        <Button Style="{StaticResource ExterminateAppBarButtonStyle}"
                Command="{Binding DeleteAllPhotosCommand}" />
      </StackPanel>
    </Grid>
  </AppBar>
</Page.BottomAppBar>
```

App bar with background color

Rotate

Separator

Like

Dislike

Right panel

Exterminate

As you can see, the app bar is just a content control that contains a single element, typically a panel. If you want to align buttons to the left or right, put them in `StackPanel` panels and set their `HorizontalAlignment` properties appropriately.

You set a color for the background of the app bar. By default, the color will be black, but I encourage you to use a color that fits the theme of your own app. Remember, you're not trying to match system colors; you just want your app to be internally consistent. That's one advantage of full-screen apps.

If you understand how to use a button on a page, you understand how to use one on the app bar. Everything you've learned about buttons so far applies here. Other than the style of the button, and the fact it happens to sit on this thing called an app bar, there's nothing special about it.

Most app bars don't include a separator, but I've seen a few that do. I've included a home-made one here so you can see one technique of how it's done. By using a rectangle and not setting an explicit height, the rectangle will appropriately resize in relation to the actual height of the app bar.

Each of the buttons uses a style with the suffix `AppBarButtonStyle`. These are predefined and available as part of the project source code. You'll need to copy the commented-out named styles in the StandardStyles.xaml file into a location near the type of the file (so they aren't commented out) or preferably into App.xaml as I did. You can also uncomment just those entries, but that becomes a bit harder to maintain. Also, in general, I don't recommend messing with the StandardStyles.xaml file directly, because it's one of those template-provided files that may change over time. For those reasons, I just copied the styles I wanted into App.xaml. The next listing shows the styles I added to the `ResourceDictionary` section of that file.

Listing 11.5 App bar button styles in the App.xaml resource dictionary

Rotate

```
<Style x:Key="RotateAppBarButtonStyle"
       TargetType="ButtonBase"
       BasedOn="{StaticResource AppBarButtonStyle}">
  <Setter Property="AutomationProperties.AutomationId"
          Value="RotateAppBarButton"/>
  <Setter Property="AutomationProperties.Name"
          Value="Rotate"/>
  <Setter Property="Content"
          Value="&#xE14A;"/>
</Style>
```

Dislike

```
<Style x:Key="DislikeAppBarButtonStyle"
       TargetType="ButtonBase"
       BasedOn="{StaticResource AppBarButtonStyle}">
  <Setter Property="AutomationProperties.AutomationId"
          Value="DislikeAppBarButton"/>
  <Setter Property="AutomationProperties.Name"
          Value="Dislike"/>
  <Setter Property="Content"
            Value="&#xE19E;"/>
</Style>
```

```
<Style x:Key="LikeAppBarButtonStyle"                                Like
       TargetType="ButtonBase"
       BasedOn="{StaticResource AppBarButtonStyle}">
  <Setter Property="AutomationProperties.AutomationId"
          Value="LikeAppBarButton"/>
  <Setter Property="AutomationProperties.Name"
          Value="Like"/>
  <Setter Property="Content"
          Value="&#xE19F;"/>
</Style>

<Style x:Key="ExterminateAppBarButtonStyle"                            Exterminate
       TargetType="ButtonBase"
       BasedOn="{StaticResource AppBarButtonStyle}">
  <Setter Property="AutomationProperties.AutomationId"
          Value="ExterminateAppBarButton"/>
  <Setter Property="AutomationProperties.Name"
          Value="Exterminate"/>
  <Setter Property="Content"
          Value="&#x2604;"/>
</Style>
```

Each of the styles in this listing is used in the app bar. The first three—rotate, dislike, and like—are all from the styles provided with the project template. The fourth, `ExterminateAppBarButtonStyle`, is one I created. I used Character Map to find an appropriate Segoe UI Symbol glyph and then copied its character code into the `Content` property.

> **NOTE** The existing styles are commented out because you take a load-time hit for each resource you have in your app, and the team wanted to make all the common styles available to you. No app would use all the resources, but they would take the startup load hit if the resources were all present in app.xaml. Remember, Windows Store apps have strict guidelines for performance and startup time.

I used the comet symbol as the glyph for my “exterminate” button, something meaningful only if you’re a dinosaur. My first thought was to use a Dalek image as I did on my site, but images are not easy to use in app bar buttons. Tom McKearney, my tech editor, suggested I name at least one control `AdmiralAppBar`.

The existing app bar button styles use font glyphs for the images because that will give you the crispest possible rendering regardless of DPI. Using PNGs is not something supported out of the box because you need an entire grid of them to represent all the states in a single button. Toolkits such as the WinRT XAML Toolkit (http://jupitertoolkit.codeplex.com/) include specialized controls like the `ImageButton`, designed for making it easy to use images. Check your rendering results before you go this route: Unless you have the sizes and DPI spot on, the PNGs will appear fuzzy in comparison to the glyphs.

You can also find app bar button images and markup online from companies like Syncfusion (syncfusion.com), which has well over a thousand icons in its studio product. You can find others at theNounProject.com and theXamlProject.com, among many more. Some of those will even generate path language XAML, which is easier to scale.

11.2.2 Wiring with commands

Each of the app bar buttons is bound to a command. You can use event handlers, of course, but you've been using commands so far, so there's no reason to stop now. In addition to the uncluttered code-behind, one thing I really like about commands is the built-in support for enabling and disabling the buttons.

Each of the commands is exposed by the `CategoryBrowserViewModel` class. For this to work, you'll need to add those commands to the viewmodel. The next listing has the additions for that class.

Listing 11.6 `CategoryBrowserViewModel` updates with commands

```
public CategoryBrowserViewModel()
{
  DeleteAllPhotosCommand = new RelayCommand(
        () => DeleteAllPhotos(),
        () => CanDeleteAllPhotos());

  RotateSelectedPhotoCommand = new RelayCommand(
        () => RotateSelectedPhoto(),
        () => CanRotateSelectedPhoto());

  LikeSelectedPhotoCommand = new RelayCommand(
        () => LikeSelectedPhoto(),
        () => CanLikeSelectedPhoto());

  DislikeSelectedPhotoCommand = new RelayCommand(
        () => DislikeSelectedPhoto(),
        () => CanDislikeSelectedPhoto());
}

private Photo _selectedPhoto;
public Photo SelectedPhoto
{
  get { return _selectedPhoto; }
  set
  {
    Set<Photo>(() => SelectedPhoto, ref _selectedPhoto, value);

    RotateSelectedPhotoCommand.RaiseCanExecuteChanged();
    LikeSelectedPhotoCommand.RaiseCanExecuteChanged();
    DislikeSelectedPhotoCommand.RaiseCanExecuteChanged();
  }
}

public RelayCommand DeleteAllPhotosCommand
{
  get; private set;
```

Command creation

Change notification (uncommented existing code)

Delete command

```
}
```

Delete implementation

```
private void DeleteAllPhotos()
{
  if (Category != null)
  {
    Category.Photos.Clear();

    AllCategories.Remove(Category);

    Category = null;
    SelectedPhoto = null;
  }
}

public bool CanDeleteAllPhotos()
{
  return Category != null && Category.Photos != null &&
        Category.Photos.Count > 0;
}
```

Rotate command

```
public RelayCommand RotateSelectedPhotoCommand
{
    get; private set;
}
```

Rotation implementation

```
private void RotateSelectedPhoto()
{
  if (SelectedPhoto != null)
    SelectedPhoto.RotationAngle =
            (SelectedPhoto.RotationAngle + 90) % 360;
}

public bool CanRotateSelectedPhoto()
{
  return SelectedPhoto != null;
}
```

Like command

```
public RelayCommand LikeSelectedPhotoCommand
{
  get; private set;
}
```

Like implementation

```
private void LikeSelectedPhoto()
{
  if (SelectedPhoto != null)
    SelectedPhoto.LikesCount += 1;
}

public bool CanLikeSelectedPhoto()
{
    return SelectedPhoto != null;
}
```

```
public RelayCommand DislikeSelectedPhotoCommand
{
    get; private set;
}

private void DislikeSelectedPhoto()
{
  if (SelectedPhoto != null)
    SelectedPhoto.DislikesCount += 1;
}

public bool CanDislikeSelectedPhoto()
{
  return SelectedPhoto != null;
}
```

Dislike command

Dislike implementation

Each of the commands follows a similar pattern: a property for the command so the UI can bind to it, a method to perform the action, and a method that tells you if the command can be executed. The result of this last method is what enables/disables the buttons. The relationship among all of these is established in the constructor when the property is created.

If you plan to test the viewmodel outside of the commands, change the action methods (such as `LikeSelectedPhoto`) to be public rather than private.

Also, remember that commented-out line in the setter in the `Category` property of the viewmodel? Now is the time to uncomment that because the command it references now exists.

Several of the new commands rely on nonexistent properties of the `Photo` model class. The next listing includes those new properties.

Listing 11.7 Rotate and ratings support in the `Photo` model class

```
using GalaSoft.MvvmLight;
using System;

namespace PhotoBrowser.Model
{
  public class Photo : ObservableObject
  {
    public Uri ImageUri { get; set; }
    public string DisplayName { get; set; }
    public string Category { get; set; }

    private int _likesCount = 0;
    public int LikesCount
    {
      get { return _likesCount; }
      set
      {
        Set<int>(() => LikesCount, ref _likesCount, value);
      }
    }
```

New base class

Existing properties

LikesCount property

```
    private int _dislikesCount = 0;
    public int DislikesCount
    {
      get { return _dislikesCount; }
      set
      {
        Set<int>(() => DislikesCount, ref _dislikesCount, value);
      }
    }

    private double _rotationAngle = 0.0;
    public double RotationAngle
    {
      get { return _rotationAngle; }
      set
      {
        Set<double>(() => RotationAngle, ref _rotationAngle, value);
      }
    }
  }
}
```

DislikesCount property

RotationAngle property

The commands themselves simply modify these new properties, so there's not much to understand there. The UI manifestation of the results is the interesting part. Figure 11.3 shows the rating (it's over 9000!) and the rotation of the image.

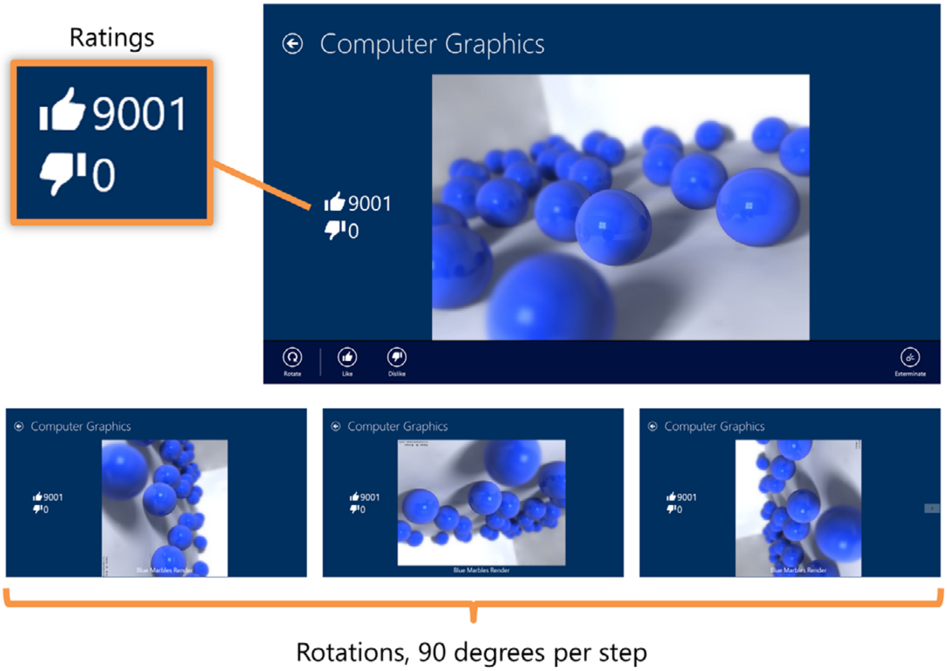

Ratings
9001
0
Computer Graphics
9001
0
Rotate
Like
Dislike
Exterminate
Computer Graphics
9001
0
Blue Marbles Render
Computer Graphics
9001
0
Blue Marbles Render
Computer Graphics
9001
0
Blue Marbles Render
Rotations, 90 degrees per step

Figure 11.3 **Ratings and rotation in use in the app. Note that the rotation uses a render transform, so it doesn't resize to fit the page. See the "Render transforms" sidebar for more information. I rendered these blue marbles in TrueSpace years back; the radiosity calculations alone took most of the day.**

The following listing includes the XAML to implement the new rotation, Likes, and Dislikes UI features based on the new data in the `Photo` model class.

Listing 11.8 CategoryBrowserPage XAML rotate and ratings support in the `FlipView`

```
<FlipView ItemsSource="{Binding Category.Photos}"
          SelectedItem="{Binding SelectedPhoto, Mode=TwoWay}"
          Padding="116,0,40,0">

  <FlipView.ItemTemplate>
    <DataTemplate>
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="*" />
          <RowDefinition Height="50" />
        </Grid.RowDefinitions>

        <Image Grid.Row="0"
               RenderTransformOrigin="0.5,0.5"
               Stretch="Uniform">
          <Image.Source>
            <BitmapImage UriSource="{Binding ImageUri}" />
          </Image.Source>

          <Image.RenderTransform>
            <RotateTransform Angle="{Binding RotationAngle}" />
          </Image.RenderTransform>
        </Image>

        <TextBlock Grid.Row="1"
                   Foreground="White"
                   Margin="5"
                   TextAlignment="Center"
                   TextWrapping="Wrap"
                   FontSize="28"
                   Text="{Binding DisplayName}" />

        <StackPanel Grid.Row="0"
                    HorizontalAlignment="Left"
                    VerticalAlignment="Center"
                    Orientation="Vertical">
          <StackPanel.Resources>
            <Style TargetType="TextBlock">
              <Setter Property="FontFamily"
                      Value="Segoe UI Symbol" />
              <Setter Property="FontSize"
                      Value="42" />
            </Style>
          </StackPanel.Resources>

          <StackPanel Orientation="Horizontal"
                      HorizontalAlignment="Left">
            <TextBlock Text="&#xE19F;" />
            <TextBlock Text="{Binding LikesCount}" />
          </StackPanel>
```

SelectedItem binding

Transform origin at midpoint

RotateTransform

Like/Dislike display style

Likes display

```
          <StackPanel Orientation="Horizontal"
                      HorizontalAlignment="Left">
            <TextBlock Text="&#xE19E;"/>
            <TextBlock Text="{Binding DislikesCount}"/>
          </StackPanel>
        </StackPanel>

      </Grid>
    </DataTemplate>
  </FlipView.ItemTemplate>
</FlipView>
```

Dislikes display

The Likes and Dislikes displays here use simple binding to get to properties of the `Photo` model class.

The rotation is a UI-only feature; you're not actually doing anything with the bits in memory or on disk. In fact, because the main page doesn't use a render transform for displaying its images, the image will not appear rotated there. For more information on render transforms, such as the one used here, please see the "Render transforms" sidebar.

Render transforms

Listing 11.8 uses a `RenderTransform`, a `RotateTransform` to be specific, to rotate the image. A `RenderTransform` is a way to modify UI elements at render time, without affecting their layout.

In chapter 5, you learned how the layout is calculated for a given element. Render transforms happen after the layout is calculated, so a resize, move, or similar transform neither causes an additional layout pass nor changes the layout slot allocated to the element.

Because of this, render transforms perform really well, especially when animated. The downside, as you'll see when you rotate the image in this section, is you don't get any of the layout magic required to fit an element inside a specific space.

The currently supported render transforms are `RotateTransform` to rotate an element (what we use here), `ScaleTransform` to change an element's size, `SkewTransform` to apply a skew to the element (top is offset from the bottom, creating a parallelogram shape), `TranslateTransform` to move an element, `MatrixTransform` to use matrix math to apply multiple transforms as a single calculation, and `TransformGroup` to apply multiple transforms using discrete transforms.

For more information on render transforms, check out http://bit.ly/WinRTRenderTransform. You can also look at any of the great Silverlight content on the web or my Silverlight 5 book *Silverlight 5 in Action* (Manning, 2012), because WinRT XAML render transforms are functionally identical to what we have in Silverlight.

If you ran the app, you may have noticed that the rotation and Likes/Dislikes changes made in the category browser page don't carry over from page to page during navigation. Similarly, deleting images didn't seem to do anything useful. That's because the main page makes a call to the image service to reload the images each time, overwriting the changes. There are a number of ways to solve this, but the one I prefer is to have the image service cache the results of the call. The following listing includes the updated `ImageService` class code.

Listing 11.9 `ImageService` improvements

```
using PhotoBrowser.Model;
using System;
using System.Collections.Generic;
using System.Linq;

namespace PhotoBrowser.Services
{
  public class ImageService
  {
    private static ImageService _current;
    public static ImageService Current
    {
      get
      {
        if (_current == null)
          _current = new ImageService();

          return _current;
      }
    }

    private List<Photo> _photos;
    public IList<Photo> GetPhotos()
    {
      if (_photos == null)
      {
        string[] fileNames = new string[]
        {
          "Autumn Trees 2007.jpg", "Nature",
          "Beetle on Azalea.jpg", "Nature",
          "Blue Marbles Render.jpg", "Computer Graphics",
          "Christmas Silver Blue Glitter Poinsettias.png", "Christmas",
          "Christmas Tree Butterfly Fruit.jpg", "Christmas",
          "Christmas Under the Sea Tree Purple.jpg", "Christmas",
          "Cucumber Beetle on Pumpkin.jpg", "Nature",
          "Fall Crisp Blue Lake.png", "Nature",
          "MFOS Synth Board 2.png", "Electronics",
          "MMID 302 Union Bridge Dec 15, 2004.png", "Trains",
          "Nebula.jpg", "Computer Graphics",
          "Plasma Chamber.jpg", "Computer Graphics",
          "Shadowed Pumpkin.jpg", "Nature",
          "Synth Background.jpg", "Electronics",
          "Synth Panel.jpg", "Electronics"
        };

        _photos = new List<Photo>();
```

LINQ for RemoveAll function

Cached list

Cache creation

```
        for (int i = 0; i < fileNames.Length; i += 2)
        {
          string name = fileNames[i];
          string category = fileNames[i + 1];

          _photos.Add(
            new Photo()
              {
                ImageUri = new Uri("ms-appx:/Pictures/" + name),
                DisplayName = name.Substring(0, name.LastIndexOf('.')),
                Category = category
              });
        }
      }

      return _photos;
    }

    public void RemoveImagesWithCategory(string category)
    {
      if (_photos != null)
        _photos.RemoveAll(s => s.Category == category);
    }

  }
}
```

New function to remove images

Now, the final change is to modify the `CategoryBrowserViewModel` one last time to use the new `ImageService` method. Here's the updated `DeleteAllPhotos` method.

Listing 11.10 Updated `DeleteAllPhotos` in `CategoryBrowserViewModel`

```
private void DeleteAllPhotos()
{
  if (Category != null)
  {

    Category.Photos.Clear();

    ImageService.Current.RemoveImagesWithCategory(Category.Category);

    AllCategories.Remove(Category);

    Category = null;
    SelectedPhoto = null;
  }
}
```

Call to new ImageService method

Now when you run the app and exterminate an entire category, that change will be reflected on the main page. Similarly, setting a rating or rotating an image will stick. Because you're not doing anything in any permanent storage, simply restart the app to get the images back.

5-star ratings

To display ratings using the popular 5-star format, check out Tim Heuer's Callisto project on GitHub (https://github.com/timheuer/Callisto).

He includes a 5-star ratings control, which would work well for displaying the Likes/Dislikes percentage for the photos in this example.

To make it work, you will need to provide an additional property, possibly in the `Photo` class but probably on the viewmodel, which adds up the number of Likes and Dislikes on the selected item and then scales that result on a 0–5 scale. For example, if you had 10 Likes and 10 Dislikes, the result would be 2.5. If you had 30 Likes and 20 Dislikes, the result would be 3; 40 Likes and 0 Dislikes would be 5.

You'd then add the ratings control to the item template and bind it to the new property.

11.2.3 *Visibility and pinning*

The app bar is displayed when the user swipes up from the bottom, down from the top, right-clicks with the mouse, hits the Windows key + Z, or hits the app key on the keyboard (right-hand side of the keyboard, usually looks like a context menu). In some apps, however, it makes sense for the app bar to be visible by default. Consider a painting app where there's no default brush, for example. In that case, you'll want to have the app bar open when the app starts up. In other cases, you want the app bar not only to be open but also to be "sticky."

An app bar is sticky when it isn't dismissed simply by selecting a command; the user must actively swipe the app bar out of the way. The automatic dismissal behavior is called a "light dismiss" because the user doesn't need to click a specific element, such as a close box, or perform a specific gesture to cause the behavior to happen.

The next listing shows how to make the bottom app bar in your app be both open and sticky. Note that you don't need to use the two properties together, but it's a common usage.

Listing 11.11 Open and sticky app bar

```
<Page.BottomAppBar>
  <AppBar Background="#FF001040"
              IsOpen="True"
              IsSticky="True">
    ...
  </AppBar>
</Page.BottomAppBar>
```

Open by default

Require active dismissal

You can programmatically close the app bar by setting `IsOpen` to `False`. In addition, you can wire up event handlers to tell when the app bar has been opened or closed, in case you need to perform initialization, pause the game, or otherwise execute code when the app bar state changes.

The app bar is a simple control but a powerful UI concept. The built-in control has all the animations and behaviors you need to integrate into the platform. Within that, you're free to add anything you want to the app bar, although app bar-style buttons tend to be the most common controls. Because of that, the Visual Studio project templates include standard styles for the basic app bar button, as well as specialized styles for common buttons. Plus, because these are just buttons, you can use everything else you've learned about working with them, including commands and event handlers, to provide the link between the UI and code.

Everything you've done so far has been on the bottom app bar. If you implement only one, you'll generally make that the bottom bar. The top app bar has some interesting uses as well, which we'll investigate next.

11.3 Top app bar for navigation

Anything you can do in the bottom app bar you can also do in the top app bar—they're the same control. When deciding what to put in the top or bottom app bar, consider the logical grouping of your controls and whether something might be more usable when sliding down from the top versus sliding up from the bottom.

In general, the top app bar is used for larger items, tools, and navigation. The top app bar is sometimes called the navigation bar, especially in earlier documentation. In this context, I mean navigation between data items, not between actual pages in the app, although you could use it that way if you wanted to. Consider the Internet Explorer example. The top app bar is used to navigate between tabs within the app. Each tab is an instance of the same hosting app page but with different data (the HTML/JS/CSS).

The current workflow requires you to go back to the main page in order to pick a different category. It would be nice if you could do everything right from the category browser page. To facilitate that, you'll use the top app bar to navigate between categories without leaving the category browser page.

Figure 11.4 shows how the app looks with the navigation bar in place.

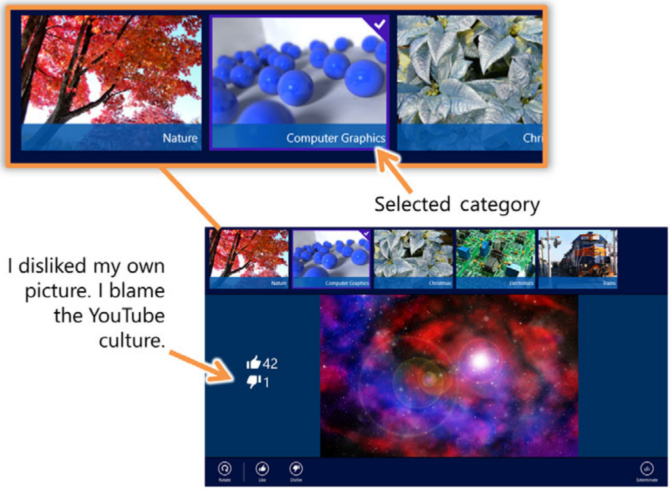


Figure 11.4 The new top app bar being used for navigation, with the currently selected category highlighted. The control used is a `GridView` with a template similar to what was used on the main page for the Semantic Zoom zoomed-out view.

The top app bar will use a `GridView`, with selection enabled, with a `StackPanel` containing image thumbnails. The overall pattern is similar to what was used in the `SemanticZoom` control in MainPage.xaml, but with elements sized to fit the top app bar.

The next listing shows the top app bar with a `GridView` used for navigation between categories.

Listing 11.12 Top app bar with navigation elements

```
<Page.TopAppBar>
  <AppBar Background="#FF001040" Height="200">
    <GridView ItemsSource="{Binding AllCategories}"
              SelectedItem="{Binding Category, Mode=TwoWay}"
              SelectionMode="Single">
      <GridView.ItemsPanel>
        <ItemsPanelTemplate>
          <VirtualizingStackPanel Orientation="Horizontal"
                                  HorizontalAlignment="Left"/>
        </ItemsPanelTemplate>
      </GridView.ItemsPanel>

      <GridView.ItemTemplate>
        <DataTemplate>
          <Grid>
            <Grid.RowDefinitions>
              <RowDefinition Height="140" />
              <RowDefinition Height="35" />
            </Grid.RowDefinitions>
            <Image Grid.Row="0" Grid.RowSpan="2"
                   Stretch="UniformToFill">
              <Image.Source>
                <BitmapImage UriSource="{Binding Photos[0].ImageUri}"
                             DecodePixelHeight="175"/>
              </Image.Source>
            </Image>

            <Rectangle Grid.Row="1"
                       Fill="#FF0055AA"
                       Opacity="0.85"/>

            <TextBlock Grid.Row="1"
                       Foreground="White"
                       VerticalAlignment="Center"
                       Margin="5"
                       TextAlignment="Right"
                       TextWrapping="NoWrap"
                       TextTrimming="WordEllipsis"
                       FontSize="15"
                       Text="{Binding Category}" />
          </Grid>
        </DataTemplate>
      </GridView.ItemTemplate>
    </GridView>
```

AllCategories binding

Two-way SelectedItem binding

Single selection mode

Horizontal stack panel

Image thumbnail

```
  </AppBar>
</Page.TopAppBar>
```

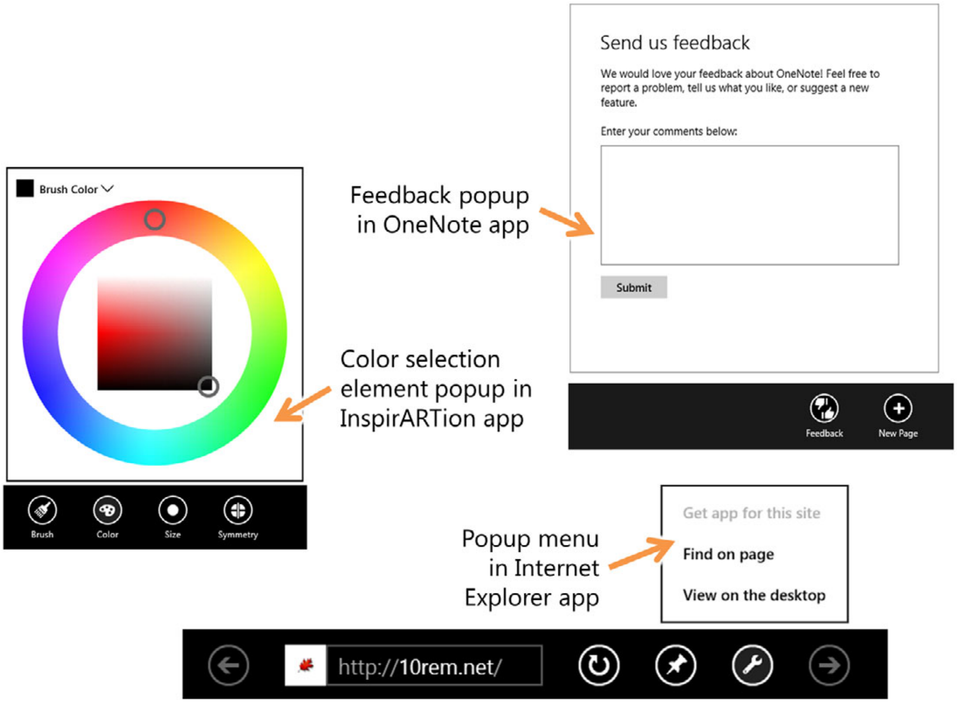
Because selection is enabled in the `GridView`, and the `SelectedItem` is bound to the `Category` property of the viewmodel, the navigation works. All the plumbing you had to do to support property change notification on the category in the original version spills over to this version, making it very easy to implement.

Sometimes you need more than an app bar can provide. For those times, you'll want to consider popups and app bar-delivered menus.

11.4 App bar popups and menus

Although simplicity is king for Modern UI apps, there'll come a time when you need to have more options than you can reasonably fit on the app bar without ruining the experience. Some apps display palettes of controls. Others, like Internet Explorer, do this by popping up menus and whole selection elements. Examples of both are shown in figure 11.5.

How you implement the popups is important, because you need to ensure they work equally well with touch and the mouse. In this section, I'll show you how to implement



Send us feedback

We would love your feedback about OneNote! Feel free to report a problem, tell us what you like, or suggest a new feature.

Enter your comments below:

Feedback popup in OneNote app

Color selection element popup in InspirARTion app

Popup menu in Internet Explorer app

http://10rem.net/

Figure 11.5 Three examples of popups from app bar buttons. OneNote (beta) used a popup to enable you to submit feedback, InspirARTion uses it for several selection tasks, and Internet Explorer uses it for the Tools menu.

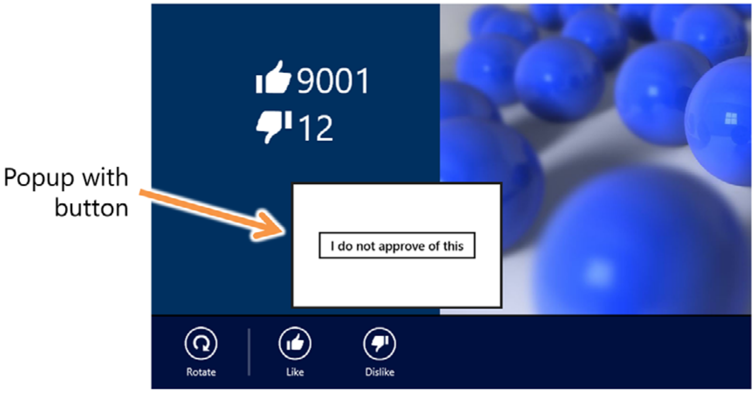


Figure 11.6 The popup launched when the Dislike button is pressed

popups, taking into account the "light dismiss" behavior required to be touch friendly. The end result will be a popup with a single button, as shown in figure 11.6.

Place the XAML in the following listing inside the main `Grid`, just after the `Grid.RowDefinitions` property.

Listing 11.13 A popup element with a single button

```
<Popup x:Name="DislikePopup"
       Width="250" Height="150"
       Margin="165,0,0,95"
       HorizontalAlignment="Left"
       VerticalAlignment="Bottom"
       Grid.Row="0"
       Grid.RowSpan="2"
       IsLightDismissEnabled="True">
  <Grid Width="250" Height="150" >

    <Rectangle Fill="White"
               Stroke="#FF202020"
               StrokeThickness="3" />

    <Button Content="I do not approve of this"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Foreground="Black"
            BorderBrush="Black"
            Command="{Binding DislikeSelectedPhotoCommand}" />
  </Grid>
</Popup>
```

Light dismiss enabled

Background

Standard button

Command binding

This markup creates a `Popup` with the `Grid` as its contents. Inside the `Grid`, you use a white rectangle as the background and a regular button as the only interactive element.

Once you have the `Popup` in XAML, you need to make a change to the Dislike button to show the popup rather than call the viewmodel method. In the CategoryBrowserPage.xaml file, modify the Dislike button in the app bar so that it no longer uses a command but instead fires off an event. The next listing shows the change.

Listing 11.14 Updated bottom app bar button

```
<Page.BottomAppBar>
  <AppBar Background="#FF001040">
...
    <Button Style="{StaticResource DislikeAppBarButtonStyle}"
            Click="OnDislikeClick" />
...
  </AppBar>
</Page.BottomAppBar>
```

Button with event handler

I chose to change the interaction to an event handler because I considered this a UI implementation detail, not something functional. The event handler for the Dislike button is shown next. Its sole responsibility is to display the popup.

Listing 11.15 Code to display the popup in MainPage.xaml.cs code-behind

```
private void OnDislikeClick(object sender,
                            Windows.UI.Xaml.RoutedEventArgs e)
{
  DislikePopup.IsOpen = true;
}
```

Display popup

You display the popup by setting its `IsOpen` property to true. Once this code is in place, you can run the app and click the Dislike button. The popup will display. You can click the button as many times as you want, but as soon as you click outside the popup, it will be dismissed. That is a result of the light dismiss behavior enabled with `IsLightDismissEnabled="true"` in the XAML.

Popups are very easy to create. But should you want to use a prebuilt menu control that's even easier to use, Tim Heuer has one in his Callisto Windows 8 toolkit on GitHub.

11.5 Summary

The app bar is one of the most important controls for Windows 8 apps. With the exception of apps with very specialized UIs (such as games and some creative apps), you'll find them in almost every app. They are the Modern UI equivalent of the menus and toolbars you've used on the desktop.

There are two app bars: the top app bar and the bottom app bar. In your own apps, you can put anything you'd like in the app bars. Don't let my buttons and navigation examples limit you. Instead, look to see what other apps are doing, and see if the UX works for you. You have a lot of freedom to be creative with how the features are used, as long as you're putting the user first, putting them in control, and providing a great user experience.

Buttons on the app bar work just like buttons do anywhere else in XAML. You can use event handlers with them, and you can wire up commands and viewmodels. If you want an app bar-specific look, you can use the built-in styles to provide the correct look and feel.

The app bar is definitely more space constrained than the toolbars of old. In some cases, you'll need to provide popups, flyouts, or other UI elements to extend the available surface area for controls. Before doing so, think hard about whether you're trying to accomplish too much on the same screen. Once you're sure the complexity is reasonable, go ahead and use the `Popup` control to create your own UI extensions.

In the next chapter, we'll further this example and tackle some of the other common UI elements that every app can use.

12

The splash screen, app tile, and notifications

This chapter covers

- Splash screens
- Extending the splash screen
- Static and live tiles
- Notification toast

As I sit here writing this book on my main PC, my remote desktop session into one of my Windows 8 tablets is showing me a colorful Start page, full of information. I can see that I have unread email and that a friend's birthday is tomorrow. I can see that my wife commented on a photo of mine on Facebook, and I can see the rather humorous label my son gave to a picture in his workbook. I can also see that my next possible achievement in Minesweeper will be "Savior of the world."

All these apps are providing information to me, beckoning me to use them, to crack them open and consume the bits. The description of the Start page sounds, perhaps, a bit busy or even distracting, but it isn't. The presentation of this information is uniform. The animations aren't too crazy, and most important, the information is useful.

Even the tiles that aren't animating are providing useful info. I can see the temperature is 52 degrees outside right now. This is important, because my current home office is in the basement corner, in a room with no windows. It's how I maintain my authentic computer-geek, milky-white complexion.

And just now, while typing this, I received a notification that I have a new message. These notifications, and the static and live tiles that make up the Start page, are a core part of the Windows experience. Learning how to use them effectively is as important as figuring out the navigation, app bar, or any other part of your app's user experience.

When I clicked on that notification, I was presented with a quick splash screen before the app came up and was ready to use. Although the splash screen is much more transient than the tile, it's another important part of the user experience.

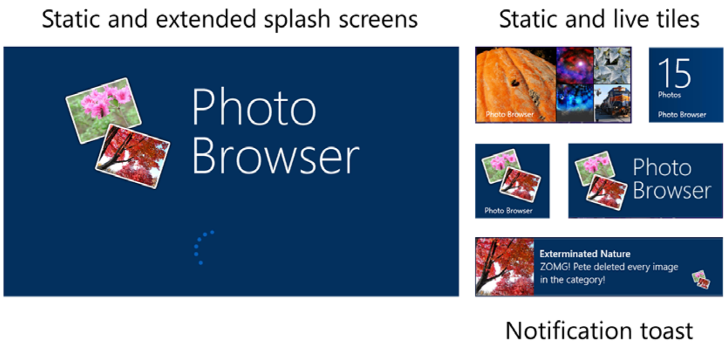
On the desktop, we've never had good enforced standards for notifications, for tile information (desktop gadgets maybe, task bar icon overlays, possibly), and for splash screens. Most applications implemented them any way they saw fit. Windows Store apps have good standards and frameworks for all of these things; that's what this chapter is about. Figure 12.1 shows the things we'll add to the PhotoBrowser app we've been working with for the past few chapters.

We'll start with a look at the humble splash screen and quickly move on to its big brother, the extended splash screen. If you're looking for a way to load data or perform another long-running task on startup from managed code, the extended splash screen is what you want.

Once we wrap up the splash screen, we'll take a small step backward user experience–wise and look at the Start page tiles. Much like the splash screen, we'll start with the simplest static versions. Then, we'll pin secondary tiles that enable deep linking into the app. Once we have all of the static tile work covered, we'll kick it up a notch and start working with live tiles. We'll implement several different types of tile notifications, including queuing up several pending notifications so they cycle through an animation on the Start page.

The final thing we'll cover in this chapter is notification toast. Tiles are great when you're staring at the Start page, but when you're in another app, or a different page in the same app, or even on the desktop, toast notifications are the best way to get the user's attention.

Everything in this chapter will continue to build on the PhotoBrowser app from the previous chapters. Let's start by adding a proper splash screen.



Exterminated Nature
ZOMG! Pete deleted every image in the category!

Figure 12.1 The splash screen, app tiles, and notifications for the PhotoBrowser app

12.1 Splash screens

You never get a second chance to make a first impression, right? Well, for your app the first impression of it running is the splash screen. This is what the user sees when they click your app's tile and launch the app.

The purpose of the splash screen is to provide a transition from the Start page to your app. It's there to show the user that your app is, in fact, launching and not crashed or stuck. By using colors that fit your app's color scheme, the splash screen also provides a visual transition from the multicolored Start page into your well-designed app UI.

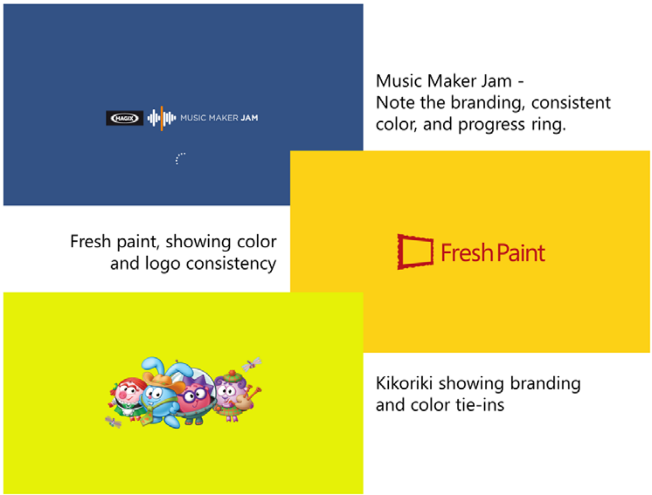
The simplest splash screen is composed of an image and a background color. This is displayed by Windows when loading your app. Once the app is loaded, your first screen is activated and you're ready to go. But sometimes you need to do more up-front initialization. Maybe you need to calculate samples or load images. For those times, you can extend the splash screen seamlessly so that the user never even realizes a second splash screen was loaded.

In this section, you'll first create a good static image and color for your splash screen. Then, because app initialization often must happen up front, you'll extend the splash screen to display a progress ring and to perform potentially time-consuming but critical work.

12.1.1 The static splash screen

The splash screen is the entire full screen that's shown when your app is loading. It consists of an image and a background color.

Size-wise, the static splash screen image needs to be 620 x 300 pixels, landscape mode. When you think about the minimum tablet resolution of 1366 x 768, the splash screen image takes up only about 18% of the pixels at the minimum resolution, as shown in figure 12.2. Because the background is such a large part of what the user sees, picking the right background color is important.



Music Maker Jam -
Note the branding, consistent
color, and progress ring.

Fresh paint, showing color
and logo consistency

Kikoriki showing branding
and color tie-ins

Figure 12.2 Three example splash screens, each showing branding and color consistency with their apps. Notice how much of the specified background color shows up around the splash screen. Picking the right background color is just as important as picking the right image. Also, notice how the Music Maker Jam app has a loading animation right on the splash screen.

To create a basic splash screen for your app, you need to do two things:

- Design an image and background color.
- Configure the appx manifest with those values.

The design of the splash screen image should be simple and should include obvious name or brand information. It can be multicolor but should have a background that's either transparent or blends in with the background color picked as part of the design. In both cases, a PNG will serve you better than a JPG, which can be neither transparent nor good for exact color matching.

> **TIP** Be sure to create the splash screen in the three DPI scaling versions—100%, 140%, and 180%—as discussed in chapter 7. This way, you'll be prepared for higher DPI displays without your app looking fuzzy. Don't get trapped in the position Android and iOS developers have been in where they have to constantly redo their images to keep up with advances in resolution. Microsoft has made it easy for you to get it right from the start, as long as you use the capability.

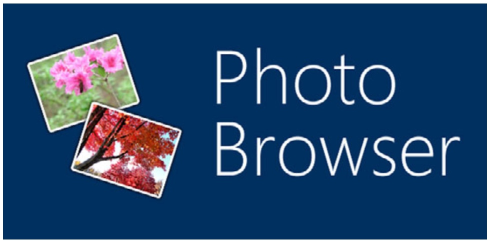
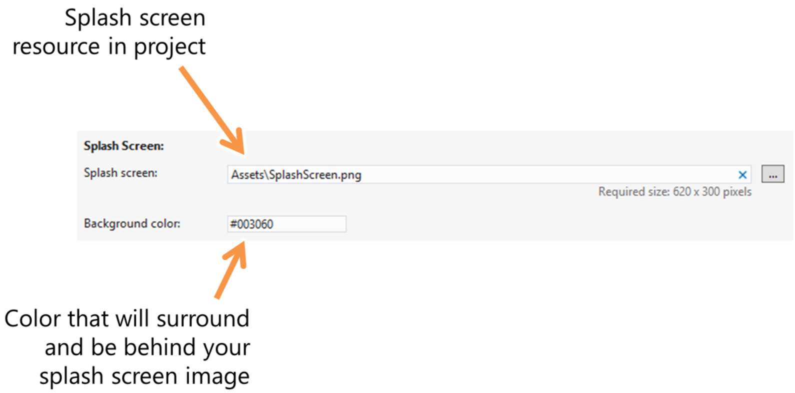


Figure 12.3 The simple splash screen for this app

Figure 12.3 shows the image I picked for this app. I decided not to make it transparent, although doing so would give you more flexibility for picking the background color.

The image simply shows two photos and has the text PhotoBrowser done in Segoe UI Light. Once the image is created, all you need do is drop it into the Assets folder, overwriting the existing SplashScreen.png or the SplashScreen.scale-100.png, scale-140 and scale-180 files as appropriate. If you use a different name, that's fine; just change it in the manifest. The manifest is also where you'll set the color, as shown in figure 12.4.


Splash screen resource in project
Color that will surround and be behind your splash screen image

Splash Screen:

Splash screen: Assets\SplashScreen.png

Required size: 620 x 300 pixels

Background color: #003060

Figure 12.4 Setting the splash screen image and background color for the project is done through the appx manifest. If you use the different scale images, you'll still specify only the root filename here.

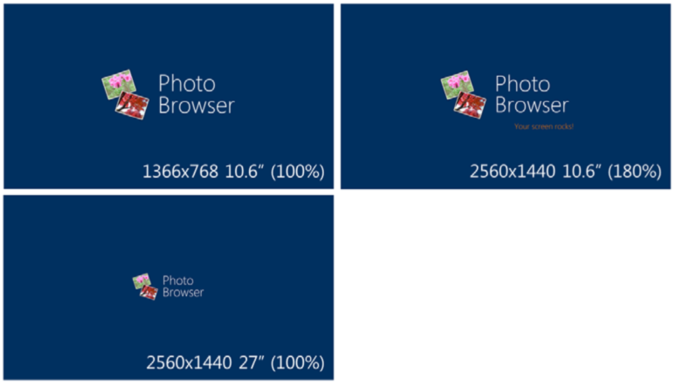


Figure 12.5 The splash screen and two different resolutions and two different screen sizes. I provided splash screens at all possible DPIs (100%, 140%, and 180%) and added a little "Your screen rocks!" message to the 180% one so you could distinguish it in print.

With both the image and color in place, you can then run the app to view the splash screen. Now is a good time to check that the colors are an exact match (print-screen the image and analyze it). What you don't want to see is a 620 x 300 box (when at 100%) in the middle, with everything else around it a different color, unless your design specifically calls for that.

You'll also want to check out the splash screen at a few common resolutions, to make sure any text or detail you have is appropriately sized. Figure 12.5 shows the final splash screen, both image and background, for this app at different resolutions. Remember, you can easily try this out with the Simulator.

With all three DPI settings supported, you'll see a crisp splash screen regardless of the user's configuration. If you had supplied only the default size, the 180% version would have been a somewhat blurry resize of your image—not a great first impression.

The splash screen is kept onscreen only long enough to allow your app to load and display the opening UI. Don't display advertising, instructions, user agreements, or anything complex or lengthy on the splash screen. It goes without saying that you should also not put in any code to deliberately keep the splash screen displayed for a minimum amount of time. The ideal scenario is a splash screen that displays for almost no time at all.

The initial display of the splash screen is handled by Windows, not your app. This is why it is displayed immediately, without any real load-time hit. But once loaded, your app can extend the splash-screen-loading process and do something useful like display load-time animation.

> **TIP** If you run the app in the Simulator and you don't see the new splash screen, manually uninstall the app from the Simulator, clean the solution, and rerun. The splash screens are sometimes cached.

12.1.2 Extended splash screens

In many cases, when an app loads and the code starts executing, it still has a lot of other work to do to restore state or otherwise prepare itself for the user. Some of these tasks may be performed in the background, while the app is running. Others must be completed before the UI is useful. It is in support of this latter scenario that we have the ability to create a second splash screen that trails the image-based version.

The extended splash screen should use the same background color as the main splash screen and should position the splash screen graphic in the same location. This will prevent any jarring transition. Beyond that, it may do anything to show progress. Typical examples are progress rings or status text. Figure 12.6 shows the progress ring used in the PhotoBrowser app.

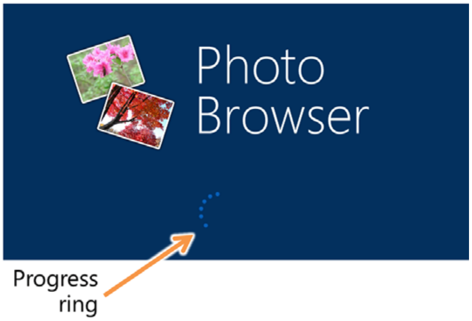


Figure 12.6 The center of the extended splash screen with the progress ring

To extend the splash screen, you use the `SplashScreen` class in concert with a dedicated splash screen page. The `SplashScreen` class provides you with the coordinates to use to display the `SplashScreen` image, as well as an event that fires when the screen is dismissed.

Before you do that, though, you need to have a reason to have an extended splash screen. Normally, this would be some long-running data loading, so you'll simulate that. Modify the `ImageService` class in the Services folder and add the function shown here.

Listing 12.1 New time-wasting function in `ImageService`

```
public Task<int> DoLongRunningInitializationAsync()
{
  return Task.Delay(5000).ContinueWith((t) => { return 42; });
}
```

Wait five seconds

You'll need to add a `using` statement for `System.Threading.Tasks` for the function to compile.

This function simply wastes a little time using the delay function of the `Task` method using a more modern and UI-friendly analog to `Thread.Sleep(5000)`. By calling this function from the splash screen code, you'll get a delay without tying up the UI thread itself. We'll talk more about the `await` and `async` keywords and asynchronous code in chapter 16.

You're about to add another page to the app, and that page must share the same background color you're using throughout. So far, you've simply hardcoded the background color onto each page. That's just not a great approach, as you learned in chapter 7 when we covered colors and resources. Let's take a moment to update App.xaml and provide a static resource with the app background color. The next listing shows how.

Listing 12.2 Creating a background color resource in app.xaml

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      ...
    </ResourceDictionary.MergedDictionaries>

    <SolidColorBrush x:Key="AppPageBackgroundBrush"
                     Color="#FF003060" />

    <SolidColorBrush x:Key="AppPageAccentBrush"
                     Color="#FF0662bf" />
    ...
  </ResourceDictionary>
</Application.Resources>
```

App-wide page background color

Accent color for ProgressRing and others

In each page in the app, update the main `Grid` background color to use this static resource using the same static resource reference as shown here:

```
<Grid Background="{StaticResource AppPageBackgroundBrush}">
```

In your project, create a new blank page named ExtendedSplash.xaml and place it in the root of the project with the other pages. The XAML contents for this screen are in this listing.

Listing 12.3 The extended splash screen page markup

```
<Page
  x:Class="PhotoBrowser.ExtendedSplash"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PhotoBrowser"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid Background="{StaticResource AppPageBackgroundBrush}">
    <Canvas HorizontalAlignment="Stretch" VerticalAlignment="Stretch">
      <Image x:Name="SplashScreenImage"
             Source="ms-appx:Assets/SplashScreen.png"
             Visibility="Collapsed" />
    </Canvas>

    <Grid x:Name="ProgressContainer"
          HorizontalAlignment="Center"
          VerticalAlignment="Top">
      <ProgressRing IsActive="True"
                    Foreground="{StaticResource AppPageAccentBrush}"
                    Margin="0,25,0,0"
                    Width="75"
                    Height="75"/>
    </Grid>
  </Grid>
</Page>
```

Container for ProgressRing

ProgressRing

The splash screen introduces a new control called the `ProgressRing`. This control is a Modern UI-style version of the indeterminate progress bar, without the usually erroneous progress value (the `ProgressBar` is also available should you need it). The ring simply shows that something is going on but doesn't indicate the percentage complete. It handles all of the animations for you, so all you need to do is activate it via the `IsActive` property.

The code-behind needs to take in an instance of the `SplashScreen` class. You'll provide this from within App.xaml.cs. For now, modify ExtendedSplash.xaml.cs so it contains the code in the following listing.

Listing 12.4 Extended splash screen code-behind

```
using PhotoBrowser.Services;
using System.Threading;
using Windows.ApplicationModel.Activation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace PhotoBrowser
{
  public sealed partial class ExtendedSplash : Page
  {
    private Frame rootFrame;
    private SynchronizationContext _uiContext;

    public ExtendedSplash(SplashScreen splashScreen)
    {
      this.InitializeComponent();

      _uiContext = SynchronizationContext.Current;

      if (splashScreen != null)
      {
        splashScreen.Dismissed += OnSplashScreenDismissed;

        Canvas.SetLeft(SplashScreenImage,
            splashScreen.ImageLocation.Left);

        Canvas.SetTop(SplashScreenImage,
            splashScreen.ImageLocation.Top);

        SplashScreenImage.Width = splashScreen.ImageLocation.Width;
        SplashScreenImage.Height = splashScreen.ImageLocation.Height;
        SplashScreenImage.Visibility = Visibility.Visible;

        ProgressContainer.Margin = new Thickness(
            0,
            splashScreen.ImageLocation.Top +
                splashScreen.ImageLocation.Height,
            0, 0);
      }
      rootFrame = new Frame();
    }

    async void OnSplashScreenDismissed(
                  SplashScreen sender, object args)
    {
      await ImageService.Current.DoLongRunningInitializationAsync();
```

Frame for navigation

Context for UI thread

Key event handler

Position, size, and show splash image

Position progress ring container

Create navigation frame

Run long running operation async

```
      _uiContext.Post((s) =>
        {
          Window.Current.Content = rootFrame;
          rootFrame.Navigate(typeof(MainPage));
        },
        null);
    }
  }
}
```

Navigate on UI thread

Setting up an extended splash screen involves moving a number of steps that are normally in App.xaml.cs over to a new splash screen code-behind file. In doing so, you're making the initialization process take a bit longer and deferring the navigation and initial page display, which normally happens in App.xaml.cs, to code you've written. The plus side here is you have complete control over when you display your main app UI and what you do during initialization.

The `SplashScreen` class provides a `Dismissed` event. This event is fired when the initial app loading has completed and the static splash is about to be removed. It's at this point, in this handler, that control is turned over to you and your extended splash screen. I handled all the UI initialization of the splash screen in the constructor of the page, so I didn't have to post any of that. All you do is call the extended initialization (your data loading, for example) and navigate to the app's main page.

You use a `SynchronizationContext` here because the `Dismissed` event handler of the `SplashScreen` is called on a different thread from the UI thread (see the sidebar). There's no explicit cross-thread exception; instead `Window.Current` is null. Note also that the event handler is marked as async. That's so you could use the `await` keyword when calling the `DoLongRunningInitializationAsync` method of the `ImageService`.

Synchronization, dispatching, and the UI thread

XAML apps have the concept of a UI thread. That's the thread where all UI controls are created and all interaction happens.

An app may have any number of additional (background) threads performing additional work. Sometimes those threads are created explicitly, using `Task.Run` or `TaskFactory.StartNew` methods, for example. Sometimes those threads are created by code external to your solution, as is the case with some of the networking code.

When a function runs on a thread other than the UI thread, the code in that function isn't allowed to manipulate UI elements. Doing so would cause any number of exceptions or odd behavior. In most cases, you'll get a cross-thread exception error. In other cases, key UI elements may be null. Any such manipulation must be dispatched to the UI thread for execution there. The two common approaches for dispatching are to use the `Dispatcher` object or the `SynchronizationContext` object.

I prefer `SynchronizationContext` because you can use it to dispatch calls to any thread, not just the UI thread, but either approach works fine. `SynchronizationContext` requires that you cache the context from the thread you wish to dispatch to. Once you do that, you can then execute code on that thread using the `Post` method.

(continued)

`Post` is a classic asynchronous method that you fire and forget. Any code that you want to have run serially on the thread must be contained within that function call.

If you're unsure if an issue is related to threading, you can display the current thread ID by calling `Debug.WriteLine` with the `System.Environment.CurrentManagedThreadId`. That value is an integer value that identifies the thread that the line of code is running on.

The `SplashScreen` instance used in the `ExtendedSplash` constructor is provided in the `OnLaunched` handler code in App.xaml.cs. Modify that function so it looks like the next listing.

Listing 12.5 App.xaml.cs `OnLaunched` function with extended splash screen

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
  if (args.PreviousExecutionState !=
          ApplicationExecutionState.Running)
  {
    var extendedSplash = new ExtendedSplash(args.SplashScreen);

    Window.Current.Content = extendedSplash;
  }
  Window.Current.Activate();
  DispatcherHelper.Initialize();
}
```

Create extended splash screen

Set content to be extended splash screen

Activate extended splash screen

This code follows the model from the original App.xaml.cs `OnLaunched` override but simplifies it a bit, by doing away with a lot of the other initialization goodies usually found here. Some of those steps, such as creating the root frame, were moved to the extended splash screen code-behind. Some of the others will be added back into the appropriate places when we cover app startup state in chapter 15.

At this point, you can run the app and see the switchover from the basic splash screen to the splash screen with the progress ring on it. After the timeout period, the app becomes visible as expected.

Once you've seen the extended splash screen a few times, you'll likely want to comment-out the call to the long-running initialization function, because it will otherwise make the rest of this chapter somewhat painful.

In your own apps, you'll need to make some hard decisions about what data to preload on startup, what data to load as needed while the app is running, and what data to load in the background while the user is using other functionality. The same applies to any long-running calculations.

Once you've made those decisions, you can see how easy it is to show progress and a splash screen while doing any preload. Even if your app doesn't have any preload work to do, a well-designed static splash screen provides an excellent transition from the Start page to the app itself.

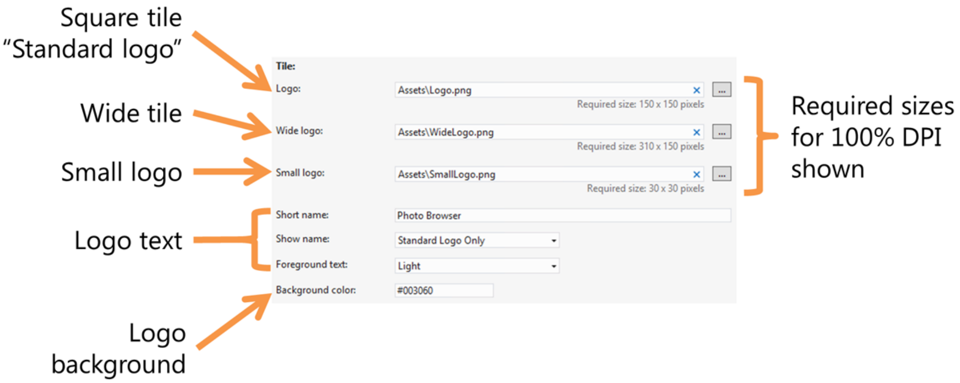
12.2 Default tiles on the start page

A default (or static) tile is the closest analog to the classic desktop application icon. It's simply a way for the user to identify and then launch your app. Once installed, it's also the very first image of your app that the user will see. Unlike the old icons, the tile can be larger and can contain a much higher-resolution image than those common 48 x 48 (and 32 x 32) icons commonly used in desktop apps. Even if you end up with live tiles, you'll always start with a base static tile.

The basic square tile at 100% DPI is a 150 x 150-pixel image. The optional wide version of that tile is 310 pixels wide by 150 pixels tall, exactly half the width and half the height of the DPI splash screen image. Just as with splash screens and other images, you should provide the square and, if used, wide tiles in the three DPI formats: 100%, 140%, and 180%. Also, consistent with everything else, you don't specify the .scale-x when referencing the image file from code, markup, or manifest.

> **TIP** If you have a desktop app, Windows will automatically pick the highest resolution icon that will fit the space allocated for the desktop icon in the tile. If you're building a desktop app, be sure to provide several different resolution icons in your resources. Windows apps can include a number of sizes up to and including 256 x 256, but it's rare to find a third-party application that provides more than the lower-resolution 32 x 32 versions.

I often use the term *tile* to mean the image. But, more correctly, the tile is the combination of the logo image, the background color, and (optionally) the foreground text. Figure 12.7 shows how to configure these options in the appx manifest designer.



Tile:

Logo: Assets\Logo.png ×
Required size: 150 x 150 pixels

Wide logo: Assets\WideLogo.png ×
Required size: 310 x 150 pixels

Small logo: Assets\SmallLogo.png ×
Required size: 30 x 30 pixels

Short name: Photo Browser
Show name: Standard Logo Only
Foreground text: Light
Background color: #003060

Figure 12.7 The tile and logo-related settings in the project's appx manifest. Only the 100% resolution sizes are shown, but the additional 140% and 180% sizes are supported and recommended. The small logo is used for certain notifications, the "all apps" view, and for uninstallation and other utility functions.

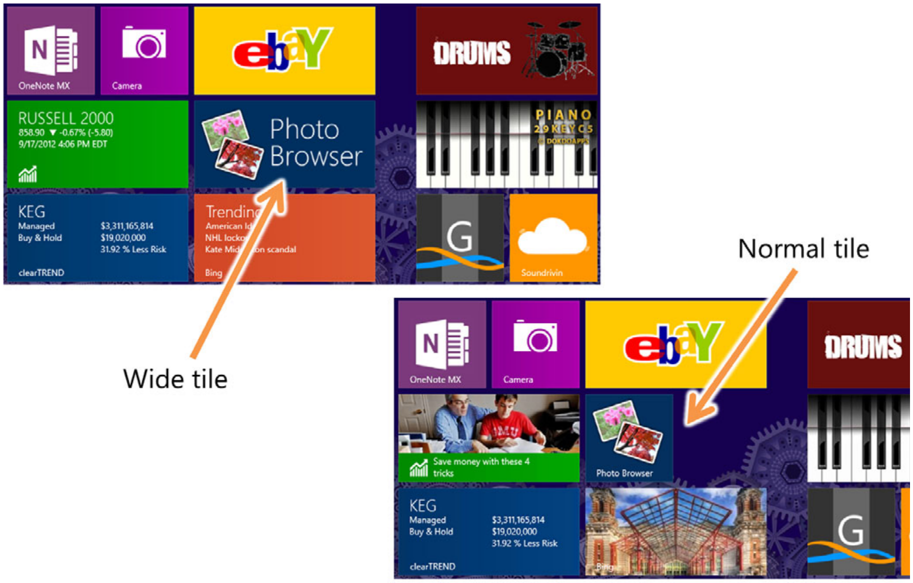


Figure 12.8 The wide and normal tiles for this app, in context. From looking at these, you may think it's better to have a bright tile background. But some Start page backgrounds are already bright, so a bright tile might get lost. Focus instead on the brand and on what you want to show.

Using the Show Name drop-down list, you can decide whether your app tile displays the app's name or just the image for both tile sizes (each may be different). For a highly branded wide tile image, you'll likely want to show the image only and incorporate the app name as part of that. For the smaller tile, or for a less highly branded wide tile, you'll want to show the app name. In figure 12.8 I have enabled the app name in the normal tile but not in the more branded wide tile.

When should you use a wide tile versus a square tile? In general, if your app is going to have frequent updates that are more than just a counter or some other tiny piece of data, you can use a wide tile. If the app does not have updates, or the updates are something as simple as a new message count, use a square tile. Although some apps do it, you shouldn't use a wide tile for static information. You can find a number of details on this design guidance on MSDN: http://bit.ly/Win8TileDesign.

You can do quite a bit more with the tile and have really good control over how it appears. We'll look at that shortly. First, I'd like to show you how to create secondary or pinned tiles for your app.

12.3 Secondary or pinned tiles

Secondary tiles are shortcuts, or deep links, into your app. Consider, for example, the situation where you have a sales or CRM (Customer Relationship Management) app, and your users typically work with a single customer for weeks or months at a time. In those cases, you may want to provide the ability to pin that customer to the Start page to avoid the navigation your app might otherwise require.

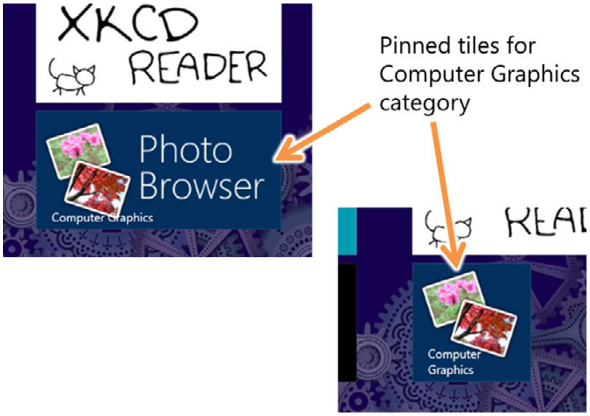


Figure 12.9 The pinned wide tile and regular tile for the Computer Graphics category. You can see that the title overlaps the image. Avoid this in your own design by providing images with space for the text.

On the Windows Desktop, any such pinning was typically a somewhat complex operation involving creating special shortcuts with command-line arguments; there wasn't much support in the UI platform for the shortcuts. Modern Windows Store apps have built-in functionality for pinning secondary styles, but it does require some work on your part; see figure 12.9.

In this section, you'll improve the app by providing the ability to pin a category to the Start page. To do this, you'll add a new button to the app bar, add a new function and command to the viewmodel, and make adjustments to the startup code from earlier in this chapter. In the end, you'll have a shortcut tile that will deep link to a category in the app.

12.3.1 Creating the tile

WinRT supplies the `SecondaryTile` class, which may be used to create the tile. You can request the tile creation from code, but the user must confirm it through a dialog box.

Each tile has a unique ID within the application. The ID of the tile is a max 64-character string beginning with a letter or number and containing alphanumeric characters. It can also contain the period or underscore, but it can't start with either of those two characters.

In addition to the ID, a tile can have parameters. The parameters are simply a string into which you can stuff name/value pairs to be retrieved when the tile is used to activate the app.

For the category pinning in this app, I created a service class to wrap most of the functionality for creating the tile and for generating IDs and parameters. The next listing shows the new `CategoryPinningService` class. Create this in the Services folder alongside the existing `ImageService` class.

Listing 12.6 Pinning functionality in the `CategoryPinningService` class

```
using PhotoBrowser.Model;
using System;
using Windows.UI.StartScreen;
```

```
namespace PhotoBrowser.Services
{
  public class CategoryPinningService
  {
    private static CategoryPinningService _current;
    public static CategoryPinningService Current
    {
      get
      {
        if (_current == null)
          _current = new CategoryPinningService();

        return _current;
      }
    }
```

Does the tile exist?

```
    public bool CategoryTileExists(PhotoCategory category)
    {
      if (category == null)
        return false;

      return SecondaryTile.Exists(
        CategoryTileIDFromCategory(category));
    }
```

Does the category match?

```
    public bool CategoryIsMatch(PhotoCategory category,
                                string sanitizedCategory)
    {
      return (bool)(SanitizeCategory(category) == sanitizedCategory);
    }
```

Clean up the category

```
    private string SanitizeCategory(PhotoCategory category)
    {
      return category.Category.Replace(" ", "");
    }
```

Create the tile ID from the category

```
    private string CategoryTileIDFromCategory(PhotoCategory category)
    {
      string id = SanitizeCategory(category);

      id = "PhotoCategory." + id;
      id = id.Substring(0, Math.Min(id.Length, 64));

      return id;
    }
```

Build a tile

```
    public SecondaryTile BuildCategoryPinTile(PhotoCategory category)
    {
      if (category != null &&
          !SecondaryTile.Exists(CategoryTileIDFromCategory(category)))
      {
```

Normal and wide images

```
        var logoUri = new Uri("ms-appx:///Assets/Logo.png");
        var wideLogoUri = new Uri("ms-appx:/Assets///WideLogo.png");

        var tileID = CategoryTileIDFromCategory(category);
```

Activation arguments

```
        var displayName = "Photos for " + category.Category;
        var shortName = category.Category;
        var arguments = "category=" + SanitizeCategory(category);

        var secondaryTile = new SecondaryTile(
          tileID, shortName, displayName, arguments,
          TileOptions.ShowNameOnLogo | TileOptions.ShowNameOnWideLogo,
          logoUri, wideLogoUri);

        secondaryTile.ForegroundText = ForegroundText.Light;

        return secondaryTile;
      }
      else
        return null;
    }
  }
}
```

The most important function in this file is the `BuildCategoryPinTile` function. This function sets up the wide and normal logos, using existing image assets. You could certainly use dedicated secondary image files if you want. The function also sets the display name for the tile (shown in the “all apps” view), the short name, and the arguments. The arguments indicate that the app is being activated from a secondary or shortcut tile.

This function does everything up to the creation of the tile but stops short of actually handling the pinning. That’s because pinning causes a dialog to appear, and that just felt wrong to have inside the service class.

The pinning itself happens on the `CategoryBrowser` page. So, the service call is wrapped into a command in the `CategoryBrowserViewModel` class. The following listing has the additions to the viewmodel.

Listing 12.7 Pinning functionality in the `CategoryBrowserViewModel`

```
public RelayCommand PinCategoryCommand
{
  get;
  private set;
}

public async void PinCategory()
{
  var tile =
    CategoryPinningService.Current.BuildCategoryPinTile(Category);

  await tile.RequestCreateAsync();
}

public bool CanPinCategory()
{
  return !CategoryPinningService.Current.CategoryTileExists(Category);
}
```

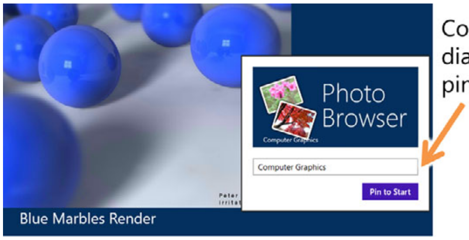
Create tile

Pin tile

In the same class, you'll need to add a `using` statement for `Windows.UI.StartScreen`. Also, be sure to add the command creation to the constructor of the viewmodel, like this:

```
PinCategoryCommand = new RelayCommand(
    () => PinCategory(), () => CanPinCategory());
```

When run (once you put in the app bar button to launch this), the app will display the confirmation dialog shown in figure 12.10 when the user clicks the app button.



Confirmation dialog for pinning

Figure 12.10 Pinning confirmation dialog. This is a system-generated dialog that uses the `SecondaryTile` information you provide.

The back-end functionality is in place. Now you need another app bar button. Luckily, from the work in the previous chapter, you already have an app bar in place on the `CategoryBrowserPage`. Because this function will work on the entire category, the button is toward the right. But because you don't want it to be confused with the button that deletes all images, you add another separator. To keep things simple, the next listing shows the entire right-aligned stack panel for the bottom app bar.

Listing 12.8 New pinning button on the `CategoryBrowserPage` app bar

```
<StackPanel Orientation="Horizontal"
            HorizontalAlignment="Right"
            Grid.Column="1">
  <Button Style="{StaticResource PinCategoryAppBarButtonStyle}"
          Command="{Binding PinCategoryCommand}" />
  <Rectangle Margin="5,15,5,15"
             Stroke="White"
             StrokeThickness="2"
             VerticalAlignment="Stretch"
             Width="1"
             Opacity="0.25"/>
  <Button Style="{StaticResource ExterminateAppBarButtonStyle}"
          Command="{Binding DeleteAllPhotosCommand}" />
</StackPanel>
```

Pin style

Command

New separator

Existing button

The app bar button relies on a style you haven't yet defined. The style is one of the stock styles, snagged from the StandardStyles.xaml resource dictionary, modified slightly, and then copied into App.xaml where the rest of the in-use app bar button styles reside. The following listing has the markup you need to add to that file.

Listing 12.9 App.xaml style for the app bar button

```
<Style x:Key="PinCategoryAppBarButtonStyle"
       TargetType="ButtonBase"
       BasedOn="{StaticResource AppBarButtonStyle}">
  <Setter Property="AutomationProperties.AutomationId"
          Value="PinAppBarButton"/>
  <Setter Property="AutomationProperties.Name"
          Value="Pin Category"/>
  <Setter Property="Content"
          Value="&#xE141;"/>
</Style>
```

Pin glyph

Now you have in place the code to create a secondary tile. At this point, you should be able to go into the app and create a tile that then appears on the Start page. Clicking it won't do anything special, however—it just brings you into the app. To make the tile really work, you need to add activation code to handle the deep linking.

12.3.2 Activating the app with the secondary tile

Right now, the app knows how to create a tile but not what to do with the tile arguments when the app is activated. You need to put in code that checks to see if any arguments were passed and, if so, whether they indicate a category parameter. Before you do that, you'll need to add in just a little infrastructure to support the new navigation model.

In previous code, the `CategoryBrowserPage` was always called from `MainPage`. Therefore, `MainPage` took care of creating and populating the viewmodel for the category page and passing it in as a navigation argument. That won't do here, so you need to add an `ImageService` function that returns the categories for you, and then you need to use that from the splash screen. The next listing includes the new `ImageService` `GetCategories` method.

Listing 12.10 `ImageService` method to retrieve categories and to match categories

```
public IList<PhotoCategory> GetCategories()
{
  var photos = GetPhotos();

  return photos.GroupBy(p => p.Category)
    .Select(p => new PhotoCategory()
    {
      Category = p.Key,
      Photos = p.ToList()
    }).ToList();
}
```

Get all photos

Group and return

The code simply replicates what was already being done in the `MainViewModel` but specifically casts to a list using `ToList`. (It could have been left as is and the return type changed to an `IEnumerable`, but I prefer returning lists.) If you want, you can update the `MainViewModel` `LoadPhotos` code to use this method rather than doing the categorization within the viewmodel.

In an app that doesn't use an extended splash screen, the activation code would go solely in App.xaml.cs. But with the extended splash screen, most of that code in our app is in the ExtendedSplash.xaml.cs file. The following listing contains the updates you'll need to support the pinned activation.

Listing 12.11 Updates to ExtendedSplash.xaml.cs to support pinned activation

```
private string _activationTileId;
private string _tileArguments;

public ExtendedSplash(SplashScreen splashScreen,
    string activationTileID, string tileArguments)
{
  this.InitializeComponent();

  _uiContext = SynchronizationContext.Current;

  _activationTileId = activationTileID;
  _tileArguments = tileArguments;

  if (splashScreen != null)
  {
  ...
  }
...
}

async void OnSplashScreenDismissed(SplashScreen sender, object args)
{
  await ImageService.Current.DoLongRunningInitializationAsync();

  _uiContext.Post((s) =>
    {
      Window.Current.Content = rootFrame;

      if (_tileArguments.StartsWith("category="))
      {
        string categoryText = _tileArguments.Substring(
                _tileArguments.IndexOf('=') + 1);

        var vm = new CategoryBrowserViewModel();

        var categories = ImageService.Current.GetCategories();

        foreach (PhotoCategory category in categories)
        {
            vm.AllCategories.Add(category);

            if (CategoryPinningService.Current.CategoryIsMatch(
                  category, categoryText))
              vm.Category = category;
        }

        rootFrame.Navigate(typeof(CategoryBrowserPage), vm);
      }
      else
      {
        rootFrame.Navigate(typeof(MainPage));
```

Class-level tile information

Parameter assignment

Check for category activation

Get category name

Create viewmodel

Load all categories

Find linked category

Navigate to category browser page

```
            }
        },
        null);
}
```

This code includes a new constructor that's used just to accept the tile arguments and ID and store them in a class variable. Most of the interesting code here is in the `OnSplashScreenDismissed` method. The older version simply navigated to the `MainPage`. This new version first checks to see if a category was provided in the arguments. If so, it then goes through the process of constructing a viewmodel, loading the categories, finding the specified category, and then navigating to the `CategoryBrowserPage`.

That's quite a bit of code in the splash screen code-behind, begging to be some place testable. If the app were to get any more complex and support multiple types of pinning, for example, I'd pull this out into a navigation-specific class. You may find that depending on how your navigation works, the extended splash screen doesn't need to be loaded when the app is deep-linked using a secondary tile. In that case, you'd create a separate navigation class that, when called from app.xaml, instantiates the appropriate pages and handles all of the initialization. As all time-constrained authors say at one point or another, I'll leave that as an exercise for the reader.

You still have a little work to do in App.xaml.cs, however. The `ExtendedSplash` screen needs to get the information from the tile activation. In this example, all you use is the arguments, but you'll pass in the `TileID` as well. Here's the updated `OnLaunched` method.

Listing 12.12 Updates to the `OnLaunched` method of App.xaml.cs

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
  if (args.PreviousExecutionState != ApplicationExecutionState.Running)
  {
    var extendedSplash = new ExtendedSplash(
      args.SplashScreen, args.TileId, args.Arguments);

    Window.Current.Content = extendedSplash;
  }
  Window.Current.Activate();

  DispatcherHelper.Initialize();
}
```

Updated constructor call

Once this code is in place, that's it. You can now create the pinned tile (or use the one you created earlier), click it, and navigate directly into a category in the app.

TIP Debugging pinned tiles takes a little extra work, because the tiles aren't pinned in the debugger, and there's no real way to simulate the arguments. There's an easy fix: Go into the project properties, Debug tab, and select the option "Do not launch, but debug my code when it starts." Then launch your app from a pinned tile, and you'll be able to step into breakpoints.

Secondary tiles are a great way to extend the experience for your app and provide additional conveniences to your users. Not only are they a way to get the user into a specific part of your app quickly, but they can also take advantages of the features provided to tiles in general, as you'll see in the next section. This provides your app with that much more surface area to get in front of the user, to encourage interaction.

Even when deep-linking, tiles are more than just static images with text. What makes tiles really special is the ability to display notification information, or to be what we like to call a "live tile."

URL-based deep links using protocol handlers

This chapter covers deep links on the Start page using pinned secondary tiles, but how would you share these links?

You can't share the secondary tiles, but you can create deep links into your app by registering as a protocol handler. This provides the ability for you to create a custom URL protocol and then handle any requests made to that protocol. The feature was really intended for apps along the lines of Skype, but it can be easily bent to allow you to provide links such as

pete://Customer/12345/invoice/57

and then mail them around. Once the link is clicked in email, your app will be launched (or activated), the `OnActivated` method in App.xaml.cs will execute, and you can parse the link and do with it what you need to do.

For more information on creating a protocol handler and activating the app, see this link: http://bit.ly/WinRTProtocolActivation.

12.4 Tile notifications or live tiles

Up until Windows 8, the only thing your app could show to the world when not running was a simple static icon. Sure, you could have a system tray icon, but your app (or some subset of it, if you architected it nicely) was still running in order to power that tray icon. Everything you've seen so far in this chapter has been a modern analog to that. What makes the Start page in Windows 8 really shine is tile notifications and the concept of live tiles.

Live tiles are tiles that show updates from the apps they represent. For example, a news live tile might show the latest headlines or images from the top stories. An even simpler example is a mail or social app that may simply show the number of unread messages as a number in the corner. Both of those scenarios require a server-side component, typically Windows Azure Mobile Services.[1] A more basic type of notification or tile is the queued live tile update and the in-app notification.

Queued live tile updates are sent to the system while the app is running but then handled entirely by the OS so that they can run while the app is closed. Notifications are a bit different in that they happen in real time, and the only way to update a notification while the app is closed is to use a server-side solution. Similarly, the only way to push live tile updates for a closed app (think of an app with new messages, tweets, or images) is to use a server-side component.

---

[1] You can learn more about Windows Azure Mobile Services at http://bit.ly/AzureMobileServices.

In this section you'll write code to send notifications that cause the tiles to update with data from the app, all on the client. First, we'll look at a simple text update to the small square tile. Then we'll look at sending combined image and text notifications. Finally, you'll learn how to queue up multiple local notifications to provide that nifty tile-flipping animation on the Start page.

12.4.1 Simple text notifications

The simplest type of notification is a single tile value, displaying dynamically created content in the form of a single notification. This type of notification can be updated any time the app is running (or from a background process if the app supports that).

Apps can send notifications to their wide tiles and to their standard square tiles. The supported types of notifications are different, so if your app supports both types of tiles, you'll need to provide separate notification templates and data for each.

Figure 12.11 shows a single tile notification used to display the number of images in the app. This is similar to showing the number of new email, unread photos, or items waiting in a queue.

To create this type of notification, you'll follow the same pattern you have elsewhere and create a `TileNotificationService` class in the Services folder. This class will be used from the `MainViewModel` to perform the notification. Create this class now using the following code.

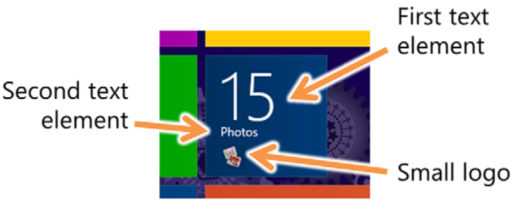


Figure 12.11 A simple text notification using the `TileSquareBlock` template with two text elements showing actual data from the app. Notice the small logo. Given its 30 x 30 size, a version without rotated images would work better here.

Listing 12.13 The `TileNotificationService` class

```
using System;
using System.Collections.Generic;
using System.Linq;
using Windows.UI.Notifications;
using Windows.Data.Xml.Dom;
using PhotoBrowser.Model;

namespace PhotoBrowser.Services
{
  public class TileNotificationService
  {
    private static TileNotificationService _current;
    public static TileNotificationService Current
  {
   get
   {
    if (_current == null)
     _current = new TileNotificationService();

    return _current;
   }
  }

  public void CreateSimpleSquareTileNotifications(
       IList<PhotoCategory> categories)
  {
   var tileXml = TileUpdateManager.GetTemplateContent(
     TileTemplateType.TileSquareBlock);

   int imageCount = ImageService.Current.GetPhotos().Count;

   var textElements = tileXml.GetElementsByTagName("text");

   var countElement = (XmlElement)textElements[0];
   var textElement = (XmlElement)textElements[1];

   countElement.AppendChild(
     tileXml.CreateTextNode(imageCount.ToString()));

   textElement.AppendChild(tileXml.CreateTextNode("Photos"));

   var notification = new TileNotification(tileXml);

   var updater = TileUpdateManager
     .CreateTileUpdaterForApplication();
   updater.Update(notification);
  }
 }
}
```

Get template XML

Get text nodes

Get count node

Get text node

Set count text

Set text

Create notification

Send notification

The format of the notification is specified using a template. The `TileTemplateType` enumeration lists all the supported templates available to the app. There are quite a few in that enumeration, each offering a different positioning of text and imagery. For the full list, see http://bit.ly/Win8TileTemplates.

Each template in the list has an associated XML template that you must fill with data. Although the schema is common across all templates, the use of the individual elements and attributes is specific to the individual templates. For the XML for each template, please see http://bit.ly/Win8TileTemplateCatalog.

In this case, there are two text nodes in the template. The first is used to display the large number on the tile. The second displays the single line of text below the large number.

Once the template values have been set, the code creates a `TileNotification` instance. This represents the actual notification to be sent to the tile. That notification is then sent to the Start page via the `TileUpdateManager` class's `Update` method.

This service needs to be called from elsewhere in the app. I decided to call it from the `MainViewModel` class's `LoadPhotos` method, right after the image collections have been populated.

The following listing has the updated `LoadPhotos` method in the `MainViewModel` class.

Listing 12.14 The `MainViewModel` `LoadPhotos` update to create notifications

```
public void LoadPhotos()
{
  var groups = ImageService.Current.GetCategories();

  Photos.Clear();

  foreach (var g in groups)
    Photos.Add(g);

  TileNotificationService.Current
      .CreateSimpleSquareTileNotifications(Photos);
}
```

Use GetCategories method

Create tile notification

If you run the app at this point, you'll be able to go back to the Start page and see the tile update with the notification. Note that if you run the app multiple times, you'll sometimes need to right-click the tile, disable live notifications, and reenable them to see the update. This happens because you haven't changed the `Tag` for the notification to make Windows see it as a new and unique notification.

One thing I don't care for in the notification is the use of the small logo. I prefer, for this app, to use the app name, as shown in figure 12.12.

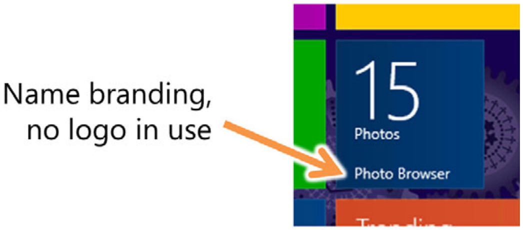


Figure 12.12 The updated notification, now using the app name rather than the app's small logo

The XML for the notification includes an attribute of the `visual` node named `branding`. This optional value can be set to `"logo"`, `"name"`, or `"none"` to control how the app's branding is displayed on the live tile. Here's the updated notification method with this change in place.

Listing 12.15 Updates to tile notification to support app name branding

```
public void CreateSimpleSquareTileNotifications(
                      IList<PhotoCategory> categories)
{
  var tileXml = TileUpdateManager.GetTemplateContent(
      TileTemplateType.TileSquareBlock);

  int imageCount = ImageService.Current.GetPhotos().Count;
```

Get “visual” node

```
  var visualElement = tileXml.GetElementsByTagName("visual")[0];

  var attribute = tileXml.CreateAttribute("branding");
  attribute.NodeValue = "name";
  visualElement.Attributes.SetNamedItem(attribute);

  var textElements = tileXml.GetElementsByTagName("text");
  var countElement = (XmlElement)textElements[0];
  var textElement = (XmlElement)textElements[1];

  countElement.AppendChild(
      tileXml.CreateTextNode(imageCount.ToString()));
  textElement.AppendChild(tileXml.CreateTextNode("Photos"));

  var notification = new TileNotification(tileXml);

  var updater = TileUpdateManager.CreateTileUpdaterForApplication();
  updater.Update(notification);
}
```

Set “branding” property to “name”

This listing does everything the prior version of this function did but with the additional logic to find the `visual` node and set its `branding` property to the value `"name"`. You’ll only know of this and other attributes if you learn the full tile schema, which can be found at http://bit.ly/Win8TileSchema.

If you run the app now, you’ll see that the tile has changed (you may need to force a refresh of it) to match figure 12.12.

Straight-text notifications are useful to keep the UI fresh with information. For more visual apps, like this photo browser, you may want to instead include images in the notifications.

12.4.2 Images in notifications

Although supported, the square tile is not often used to show images. Instead, this is where the wide tile shines. Many of the notification templates support the use of images. Some of them allow you to use several images at once to create a collage. Images used in tile notifications must be 1024 x 1024 or smaller and less than 200 KB. Most of the images supplied with this sample, and any you might substitute, will either be larger resolution-wise or larger size-wise.

CREATING THE THUMBNAIL IMAGES

We haven’t covered file IO yet, so I’ve taken an easier approach and pregenerated the thumbnails. I’ve added to the project a small thumbnail for each image in the app. The URIs for the thumbnail for each image are then made available through the `ThumbnailUri` property of the `Photo` class:

```
public Uri ThumbnailUri { get; set; }
```

Add this property to your copy of the `Photo` class in the Model folder. Once that has been done, you need to update the `GetPhotos` method of the `ImageService` class so it builds the `ThumbnailUri` as part of the image-loading process. The next listing shows the updated function.

Listing 12.16 Updated `ImageService GetPhotos` method

```
public IList<Photo> GetPhotos()
{
  if (_photos == null)
  {
...

    for (int i = 0; i < fileNames.Length; i += 2)
    {
      string name = fileNames[i];
      string category = fileNames[i + 1];

      string displayName =
              name.Substring(0, name.LastIndexOf('.'));

      string uriRoot = "ms-appx:/Pictures/";

      _photos.Add(
        new Photo()
          {
            ImageUri = new Uri(uriRoot + name),
            DisplayName = displayName,
            ThumbnailUri =
               new Uri(uriRoot + "thumbnail_" + displayName +".jpg"),
            Category = category,
            LikesCount=9001
          });
    }
  }
  return _photos;
}
```

File name without extension

Root of files URL

ThumbnailUri

To keep file sizes small, all of the thumbnails are JPEGs. The files are named like the original image, but with the .jpg extension and a prefix of thumbnail_. The code in this listing simply builds up the new `ThumbnailUri` property using this pattern. Note that if you wanted to, you could now go back to the thumbnail-generation code for the `MainPage` `GridView` as well as the navigation bar on the `CategoryBrowserPage` and update them to use the thumbnail property rather than the full-scale image. For this chapter, I didn't do that, but if you're dealing with really large image files, this can be a great way to boost performance.

GENERATING THE NOTIFICATION

For this tile notification, I decided to use the wide tile. Rather than have the entire tile taken up by a single image, I decided to use one of the collection-based templates, specifically `TileWideImageCollection`. This template lets you have one image on the left that takes up half the wide tile and four additional images arranged in a 2 x 2 grid on

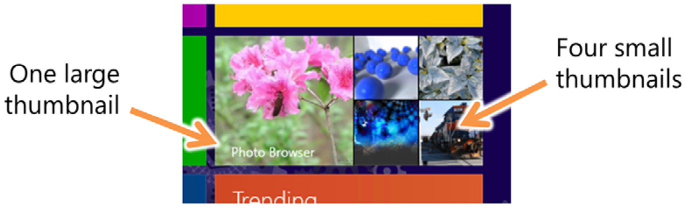


Figure 12.13 The wide tile with five image thumbnails and the app name

the right. The template itself is quite simple, having five image nodes each with an `src` attribute and an optional `alt` attribute. Figure 12.13 shows this tile for our app.

The code to generate this notification is implemented as a new method in the `TileNotificationService` class. You'll continue to keep the older notification method in there, because you want to have something that will notify the square tile as well. The new method is shown here.

Listing 12.17 New notification method in `TileNotificationService`

```
public void CreateWideTileNotifications(IList<PhotoCategory> categories)
{
  var tileXml = TileUpdateManager.GetTemplateContent(
      TileTemplateType.TileWideImageCollection);

  var visualElement = tileXml.GetElementsByTagName("visual")[0];
  var attribute = tileXml.CreateAttribute("branding");
  attribute.NodeValue = "name";
  visualElement.Attributes.SetNamedItem(attribute);

  var imageElements = tileXml.GetElementsByTagName("image");

  var rnd = new Random();

  for (int i = 0; i < 5; i++)
  {
    var element = (XmlElement)imageElements[i];
    var srcAttribute = tileXml.CreateAttribute("src");

    int categoryIndex = i % categories.Count;
    int photoIndex =
        rnd.Next(0, categories[categoryIndex].Photos.Count-1);

    srcAttribute.NodeValue =
        categories[categoryIndex]
        .Photos[photoIndex].ThumbnailUri.ToString();

    element.Attributes.SetNamedItem(srcAttribute);
  }

  var notification = new TileNotification(tileXml);

  var updater = TileUpdateManager.CreateTileUpdaterForApplication();
  updater.Update(notification);
}
```

Use name for branding

Get all image nodes

Get safe category

Get random photo

Set thumbnail source

Create notification

This code is very similar to the previous tile notification; the primary difference is which template you use. Because the template supports five images, you loop through the categories, setting the template values. You use a random image from each category and also have a modulus function in place to ensure you don't go past the number of categories in the collection.

This method then needs to be called as part of the image-loading process. In the `MainViewModel` class, in the `LoadPhotos` method, right under the previous notifications call, add the following line of code:

```
TileNotificationService.Current.CreateWideTileNotifications(Photos);
```

That's it! Run the app again, and turn the tile back into a wide tile (right-click the tile on the Start page and then select the option to turn on wide tiles). You should now see the five images displayed.

> **TIP** If you cast the `IXmlNode` for the image to an `XmlElement`, you can use `SetAttribute("src", value)` instead of the slightly longer form with `Attributes.SetNamedItem` that I used.

One thing your tile hasn't done that many on the Start page do is flip through multiple pages of data. My People app tells me about activity from my Facebook timeline, the Photos app shows me a continuous slide show of photos, and the News app scrolls through top headlines. How can you do that in your app?

12.4.3 Queuing multiple tile notifications

An app can queue multiple notifications using the aptly named notification queue. The code to create notifications is almost identical; all you need to do is enable queuing and then provide each notification with a unique tag so Windows knows each is uniquely valid and not simply an overwritten notification.

Windows will allow an app to have up to five active notifications; it ignores any beyond that (even those will eventually stop updating if the app hasn't been used for several days). The app code can overwrite existing notifications, or it can set expiration dates on notifications as they are created, thereby ensuring freshness of data over time.

As shown in figure 12.14, each of the notifications in the queue can use completely different templates. In support of that, you'll do a little refactoring of the `TileNotificationService` class and then add in support for queuing.

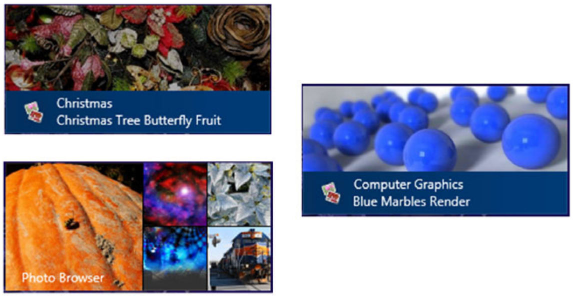


Figure 12.14 Three separate notifications using different images. Notice how the text + image template uses the app's small logo even though you explicitly set the branding to be the name. Names don't fit in that template so Windows automatically reverts to the logo.

The following listing shows the starting point for the changes.

Listing 12.18 Updated `TileNotificationService` class with queuing support

```
private void SetBrandingToName(XmlDocument tileXml)
{
  var visualElement = tileXml.GetElementsByTagName("visual")[0];

  var attribute = tileXml.CreateAttribute("branding");
  attribute.NodeValue = "name";
  visualElement.Attributes.SetNamedItem(attribute);
}

public void CreateWideTileNotifications(IList<PhotoCategory> categories)
{
  var updater = TileUpdateManager.CreateTileUpdaterForApplication();
  updater.EnableNotificationQueue(true);

  for (int i = 0; i < 5; i++)
  {
    if (i % 2 > 0)
    {
      updater.Update(CreateTileWideImageTextNotification(
          categories, "tile.t." + i));
    }
    else
    {
      updater.Update(CreateTileWideImageCollectionNotification(
          categories, "tile.i." + i));
    }
  }
}
```

Set branding to name

Enable notification queue

Alternate notifications

Generate notifications

This listing provides a new function that factors out the reusable code to set the branding to name rather than icon. The primary interface function is changed to one that simply calls out to other functions to create the five notifications. Using a simple modulus operator, you alternate between which template is used for the particular notification.

> **TIP** The tile notification SDK samples at http://dev.windows.com include a helper class designed to help reduce some of this notification code in your own apps.

There are two new tile notification methods required to support the updated queue approach. The next listing shows the first, which is the same five image tiles you created earlier.

Listing 12.19 Five-image notification method

```
private TileNotification CreateTileWideImageCollectionNotification(
    IList<PhotoCategory> categories, string tag)
{
  var tileXml = TileUpdateManager.GetTemplateContent(
        TileTemplateType.TileWideImageCollection);

    SetBrandingToName(tileXml);
```

Set branding to name

```
    var imageElements = tileXml.GetElementsByTagName("image");

    var rnd = new Random();

    for (int i = 0; i < 5; i++)
    {
      var element = (XmlElement)imageElements[i];
      var srcAttribute = tileXml.CreateAttribute("src");

      int categoryIndex = i % categories.Count;
      int photoIndex =
          rnd.Next(0, categories[categoryIndex].Photos.Count - 1);

      srcAttribute.NodeValue =
          categories[categoryIndex]
          .Photos[photoIndex].ThumbnailUri.ToString();

      element.Attributes.SetNamedItem(srcAttribute);
    }

    var notification = new TileNotification(tileXml);
    notification.Tag = tag;
```

Set unique tag

```
    return notification;
}
```

There's not much new here. The code has been factored out to set the name branding, and the method has been changed to return a `TileNotification` instance rather than adding the notification to the manager itself. Before returning the notification, you set the `Tag` property. This is what Windows uses to identify a notification and weed out duplicates.

The next listing shows a new kind of tile. This one has one wide image and two lines of text below it. You'll use that text to show the category name and then the image name.

Listing 12.20 Image and text notification method

```
private TileNotification CreateTileWideImageTextNotification(
    IList<PhotoCategory> categories, string tag)
{
    var tileXml = TileUpdateManager.GetTemplateContent(
        TileTemplateType.TileWideImageAndText02);
```

Use image and text template

```
    SetBrandingToName(tileXml);

    var imageElement = tileXml.GetElementsByTagName("image")[0];
    var categoryNameElement = tileXml.GetElementsByTagName("text")[0];
    var photoNameElement = tileXml.GetElementsByTagName("text")[1];
```

Get nodes

```
    var rnd = new Random();
    var category = categories[rnd.Next(0, categories.Count-1)];
    categoryNameElement.AppendChild(
        tileXml.CreateTextNode(category.Category));

    var photo = category.Photos[rnd.Next(0, category.Photos.Count-1)];
    photoNameElement.AppendChild(
        tileXml.CreateTextNode(photo.DisplayName));

    var srcAttribute = tileXml.CreateAttribute("src");
    srcAttribute.NodeValue = photo.ThumbnailUri.ToString();
    imageElement.Attributes.SetNamedItem(srcAttribute);

    var notification = new TileNotification(tileXml);
    notification.Tag = tag;

    return notification;
}
```

Set top text

Set bottom text

Set photo

Create notification

Set tag

The template used for this notification requires an image and two text values. For this app, the top text value is the category name; the bottom is the image name. The image itself is randomly chosen from the randomly selected category.

With this method in place, you can now run the app. Remember, you may need to disable and then enable notifications on the Start page in order to see the change. You should see up to five rotating tiles, each with random images. Now you really have a Start page presence!

Notification through live tiles is one of the best ways you can provide timely information to your user. It's a way for you to move beyond the static imagery and to instead provide information that will entice the user to come into and use the app again and again.

Tile notifications get the user into your app when they're looking at the Start page. What about cases when you need to inform the user of something when they're running another app or even another page in the same app? For those instances, you need *toast notifications.*

Lock screen notifications

Apps can also support lock screen notifications. This is similar to tile notifications but requires that you explicitly request the capability in the manifest, under the Notifications section. In addition, you must set the badge logo in the same manifest.

Beyond that, the app must run in the background, or else the lock screen notification won't be very useful or timely.

Lock screen updates build on what you've learned for tile notifications. For more information on lock screen notifications see http://bit.ly/Win8LockScreenNotifications.

12.5 Toast notifications

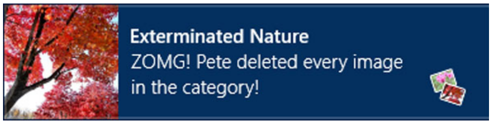
By now you've probably installed a few apps from the Windows Store. When doing so, you receive a notification on the upper right of the screen, telling you the app has completed installation. This rectangular message, containing an image and text, is called *toast*. The name is historical, from the days when Outlook would pop up new message notifications at the bottom-right corner of the screen, sliding up from the bottom like a piece of toast coming from the toaster. The name stuck.

Toast notifications are useful when your app may not have focus or the part of the app that generated the notification is no longer in focus. For example, email apps will pop up toast when another app has focus. Apps that may have to perform a long-running process server-side, like provision a site, can use toast whether or not they have focus.

Whereas tile notifications are informative and enticing, toast notifications are time-sensitive and potentially urgent, like an incoming phone call.

Notification toasts can be scheduled and recurring, like reminders coming from your favorite calendar app. They can, of course, be one-time discrete events that happen right when the app fires off the notification. The duration the notification stays visible depends on whether you use a standard or a long-duration toast. For most cases, the standard toast will be sufficient. But when you really need to grab the user's attention (phone call or appointment reminder, for example), the 25-second duration of the long-duration toast is a good choice.

The use of toast in this app will be a bit contrived but reasonable. When the user deletes the images for the category, they'll see a notification toast telling them what just happened. Figure 12.15 shows the toast notification.



Exterminated Nature
ZOMG! Pete deleted every image in the category!

Figure 12.15 The toast notification for this app. Notice the use of the small logo on the bottom right. As an aside, "Exterminated Nature" sounds more impactful than it actually was.

In this section you'll create a toast notification service class and use it to send a toast notification on behalf of the app. As a bonus, you'll use the `UserInformation` class to get the first name of the currently logged-in user to display in the notification.

> **NOTE** Desktop apps can also send toast notifications as long as they have a shortcut in the All Apps view on the Start page.

12.5.1 *Creating the notification service*

Following the same pattern as in all of the other examples, you'll create a service class to handle the actual notification. The methods of this service class will then be called from appropriate viewmodel methods in the app.

In the Services folder create a new class named `ToastNotificationService`. It will look very much like all of the other service classes. Here's the code for this class.

Listing 12.21 The `ToastNotificationService` class

```
using PhotoBrowser.Model;
using System;
using Windows.Data.Xml.Dom;
using Windows.UI.Notifications;
```

```
namespace PhotoBrowser.Services
{
  public class ToastNotificationService
  {
    private static ToastNotificationService _current;
    public static ToastNotificationService Current
    {
      get
      {
        if (_current == null)
          _current = new ToastNotificationService();

        return _current;
      }
    }

    public async void NotifyCategoryDeleted(PhotoCategory category)
    {
      var toastXml = ToastNotificationManager.GetTemplateContent
         (ToastTemplateType.ToastImageAndText02);

      var imageElement =
          toastXml.GetElementsByTagName("image")[0];

      var headingTextElement =
          toastXml.GetElementsByTagName("text")[0];

      var bodyTextElement =
          toastXml.GetElementsByTagName("text")[1];

      string heading = "Exterminated " + category.Category;
      string bodyTemplate =
          "ZOMG! {0} deleted every image in the category!";

      string userFirstName =
          await Windows.System.UserProfile
          .UserInformation.GetFirstNameAsync();

      string bodyText = string.Format(bodyTemplate, userFirstName);

      headingTextElement.AppendChild(
          toastXml.CreateTextNode(heading));

      bodyTextElement.AppendChild(
          toastXml.CreateTextNode(bodyText));

      if (category.Photos.Count > 0)
      {
        var thumbnail = category.Photos[0].ThumbnailUri.ToString();
        ((XmlElement)imageElement).SetAttribute("src", thumbnail);
      }

      var toastElement = toastXml.SelectSingleNode("/toast");
```

Get template

Build heading

Build body

Get user's first name

Set heading

Set body

Set photo

Use long duration

Set launch parameters

```
      var launchParams = "deletedCategory";
      ((XmlElement)toastElement).SetAttribute("duration", "long");

      ((XmlElement)toastElement).SetAttribute("launch", launchParams);

      var toast = new ToastNotification(toastXml);
      var notifier = ToastNotificationManager.CreateToastNotifier();
      notifier.Show(toast);
    }
  }
}
```

Create and show toast

This example creates the toast notification using a long duration and displays a message about the current user having deleted an entire category. To get the username, you use the `UserInformationClass` in the `UserProfile` namespace.

Like tile notifications, toast notifications are template-based. You can see the full list of templates in the template catalog at http://bit.ly/Win8ToastTemplateCatalog. Also, like tile notifications, toast notifications can't have additional controls on them.

Also, like tiles, toast notifications can have launch parameters associated with them. The idea is that if you have scheduled notifications, or notifications generated externally, you can provide information in the notifications that will be passed to the app on launch, just as you did with the secondary tiles. You set a simple launch parameter here, just to show how, but don't use it.

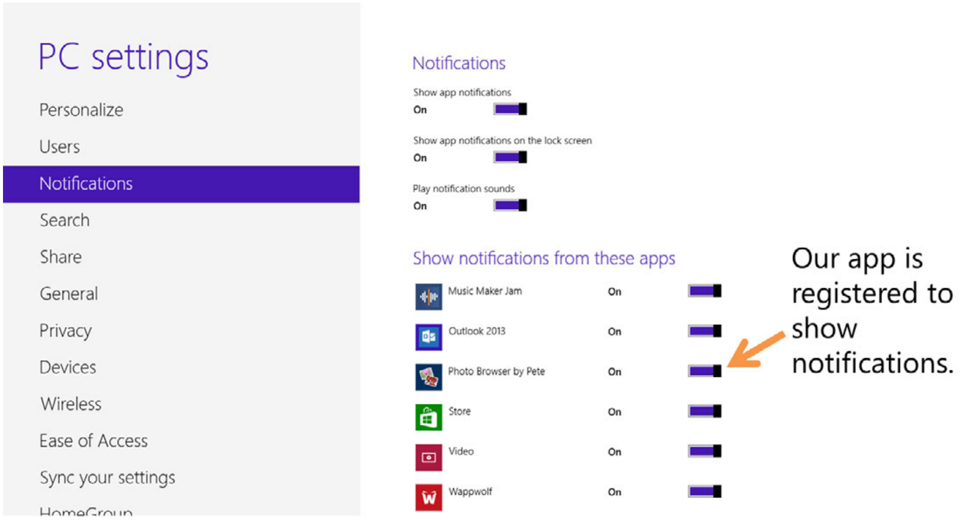
The notification code will be called from the `DeleteAllPhotos` method of the `CategoryBrowserViewModel`. That way, it will be kicked off when the user clicks the app bar button. The following listing shows the updated method, adding only the single line of code.

Listing 12.22 Updated `CategoryBrowserViewModel DeleteAllPhotos` method

```
private void DeleteAllPhotos()
{
  if (Category != null)
  {
    ToastNotificationService.Current
      .NotifyCategoryDeleted(Category);

    AllCategories.Remove(Category);

    Category.Photos.Clear();

    ImageService.Current.RemoveImagesWithCategory(Category.Category);

    Category = null;
    SelectedPhoto = null;
  }
}
```

New call to ToastNotificationService

Before you can run this app and see the toast, you must enable it.



Notifications

Show app notifications: On

Show app notifications on the lock screen: On

Play notification sounds: On

Show notifications from these apps

Music Maker Jam	On
Outlook 2013	On
Photo Browser by Pete	On
Store	On
Video	On
Wappwolf	On

Our app is registered to show notifications.

Figure 12.16 The app for this chapter shows up under the Notifications section of PC settings because it has been registered to send notifications.

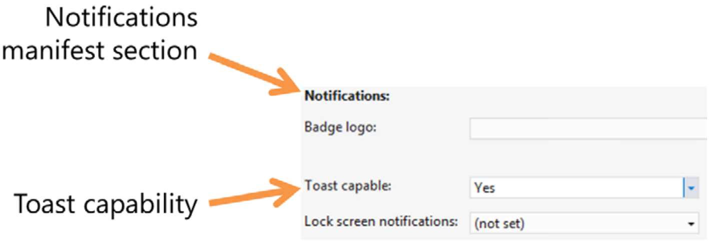
12.5.2 Enabling toast

Toast is a feature that must be specifically enabled in the appx manifest. Requesting this feature will enable toast and will also cause your app to be listed in the Notifications section of the PC settings, as shown in figure 12.16.

The appx manifest setting to enable toast notifications (shown in figure 12.17) is simply a drop-down list with Yes or No. If you don't enable this, or if the user turns off notifications, your notifications will silently fail.

With the appx manifest setting in place, you can now run the app and click the button to delete all photos in the category. While the toast notification is up, Alt-Tab to the desktop or switch to another app—you'll still see the toast notification, as long as you're within the timeout period. Click the X to close the toast, or click the toast notification itself to bring you back to the PhotoBrowser app.

Toast notification is a great way to show time-sensitive information to the user. In your own apps, you can even use it in place of the old desktop `MessageBox` for strictly informational presentation (such as "File X was saved to location Y" or "Print job completed").


Notifications manifest section

Toast capability

Notifications:

Badge logo:

Toast capable: Yes

Lock screen notifications: (not set)

Figure 12.17 Toast must be enabled in the manifest before you can use it in code.

Push notifications

All of the notifications you've seen in this chapter have been local notifications. That's great for the majority of uses, but there are times when locally generated notifications are not sufficient. Consider a situation when you need to alert the user of an event that happens on a web server (like new comments on a blog, or a completed deployment, or new items in a server-side work queue).

For those situations, notifications can also be generated from outside the PC, as push notifications from a cloud service. This is accomplished through WNS (Windows push Notification Services). For more information on push notifications and generating events from outside the PC, see http://bit.ly/Win8PushNotifications.

12.6 Summary

It could, perhaps, be easy to put this entire chapter under the "Things You Leave 'til Last" heading on an app development project. I hope I have shown you that the splash screen, tiles, and notifications are not simply the "design me an icon and splash graphic" type tasks from desktop apps but are key parts of the overall experience.

When you have no up-front loading or work to do, a simple splash screen will suffice. But if you do want to perform something time-consuming up front, using an extended splash screen is the way to go.

Your app tile is more than an icon: It's a living surface you can use to present information to the user and to encourage them to use your app. It's not the equivalent of the "Yes, we're open" sign in the store window but more like a personalized welcome banner hanging on the sidewalk, updating regularly with items of interest to you. The tile is an extremely powerful part of the user experience; make the best use of it you can.

When you want to grab the user's attention when they're in another app or on a different page in the same app, notification toast is the way to go. Toast is a standardized approach to alerting the user of events, like incoming phone calls, appointments, or other high-priority information. Do not forget, the user is always in control and can turn off the toast (or live tiles) anytime they want.

In the next chapter, we'll continue our exploration of the UI by discussing app-pinning states and how to handle orientation changes.

13

View states

This chapter covers

- View states
- The `LayoutAwarePage`
- Supporting the Snapped state
- Using Visual State Manager

If you're reading this near a desktop or laptop computer, take a look at your display. Do you have multiple overlapping windows? Do you have documents or other work side by side? I have two 30" displays on my primary PC, and I keep a ton of stuff open during the normal course of work, but rarely can I say I'm equally focused on two things at once. The closest I get is when writing these chapters, when I bounce back and forth between a remote desktop session to a Win8 tablet on one display to the Word document (and Photoshop, MSPaint, and PowerPoint) in the main display.

The reason the desktop still exists in Windows 8 is as much for power users like us to continue this type of workflow as it is for simple compatibility. Now call up the display used by your parents or your nontechnical friend. Chances are they run everything maximized and almost never run two things side by side because window management is a challenge. I remember the early days of teaching computers to folks when they would think the application "went away" when they Alt-Tabbed or otherwise moved to another app. It took many of them a long time before they realized they could simply click the icon on the taskbar and not have to relaunch the application. One time, my father had so many instances of an app open, nothing would paint; his system was completely out of GDI resources and just gave up on him.

Windows 8 Modern apps have a more standardized approach to keeping multiple apps open at the same time. You can have any number of apps waiting in the background (as long as there is sufficient memory and other resources), but you can also have two apps running side by side, sharing screen real estate. This is accomplished not using traditional windows with their inherent touch usability issues but instead by using a docking metaphor. For an app developer, these different ways to dock the app are called *view states*.

Windows 8 XAML has built-in support to help you manage view states and screen orientation. In this chapter, we'll first look at the different states and then at the considerations you have to make when supporting them in your app, especially with the app bar. We'll also adjust the UI to display different controls depending on the state. One great thing about all this is that XAML was designed to natively support a very fluid layout style, so you'll see that your app doesn't require a ton of changes, and because you've used MVVM, the changes remain isolated.

13.1 Full, filled, and snapped views

If you're an ASP.NET developer, the title of this chapter probably made you scream "Nooooo!" View states in Windows 8 apps are not to be confused with that classic ASP.NET page baggage: viewstate. Instead, view states in Windows 8 apps are the three (well, four, if you consider portrait and landscape full views) main forms an app can take when presented onscreen.

In the Windows Store, all apps must support the three views: full screen (portrait and landscape), snapped, and filled. Figure 13.1 shows the three different views, plus orientation.

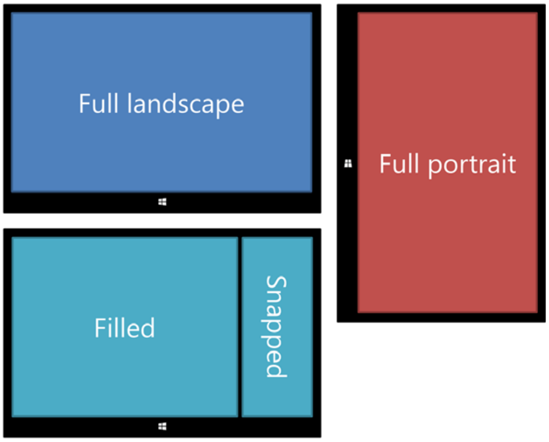


Figure 13.1 The three views for a Windows 8 Modern UI app

In order to support snapped views, the screen resolution must be at least 1366 x 768. That allows a snapped view of 320 x 768 and the filled view of 1024 x 768, plus 22 pixels for the separator bar. For those of us who speak at conferences, this minimum resolution limitation makes it difficult to demonstrate snapped states at most venues that have 1024 x 768 or 1280 x 720 projectors. In those instances, the Simulator is your best friend. If the screen resolution is below the minimum, the user cannot snap apps.

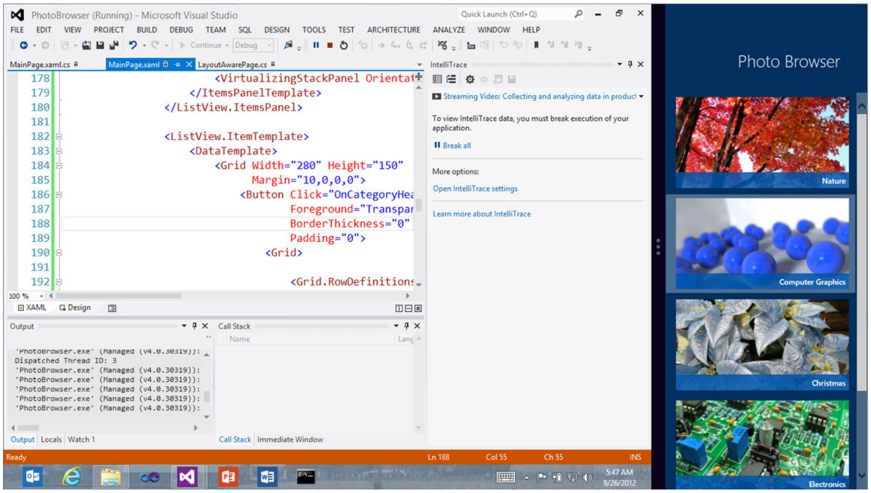
> **NOTE** The user can snap the app on the left or the right. It is entirely under their control.

Ideally, your app should remain completely functional in all views. But special consideration has been given to apps like games, which sometimes can't do anything other than pause when in snapped view. Even in those cases, I encourage you to consider doing something useful in the snapped view, such as showing the current score and leader boards, offering to continue playing the game soundtrack, or otherwise continuing to engage the user.

In pages that are based largely on a single view control, like a `GridView` or `ListView`, you may find that you don't need to do anything special to support the filled view or portrait view versus the full landscape view. In those cases, the normal XAML layout mechanisms work their magic for handling the layout changes. The work you had to do to support various-resolution displays works here. In other cases, you'll need to manually resize or rescale items to fit the layout, but you'll typically need to do that to support small screens anyway. You may tweak some margins, but that's usually about it for many apps.

The snapped view almost always requires significant layout changes. That's because it is so much smaller than the other views, smaller than the smallest supported screen resolution of 1024 x 768, and also represents a secondary focus for the user.

When an app is in snapped view, it's because the user is still interested in interacting with the app but not as their primary task at that time. A perfect example is a Twitter app


PhotoBrowser (Running) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL DESIGN TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Continue
Debug
MainPage.xaml.cs
MainPage.xaml
LayoutAwarePage.cs
IntelliTrace
100 %
XAML
Design
Output
Call Stack
Name
Lang
Output
Locals
Watch 1
Call Stack
Immediate Window
Ready
Ln 188
Col 55
Ch 55
INS
5:47 AM
9/26/2012
Photo Browser
Nature
Computer Graphics
Christmas
Electronics

```
                <VirtualizingStackPanel Orientat
            </ItemsPanelTemplate>
        </ListView.ItemsPanel>

        <ListView.ItemTemplate>
            <DataTemplate>
                <Grid Width="280" Height="150"
                      Margin="10,0,0,0">
                    <Button Click="OnCategoryHea
                            Foreground="Transpar
                            BorderThickness="0"
                            Padding="0">
                        <Grid>

                            <Grid.RowDefinitions
```

Streaming Video: Collecting and analyzing data in product

To view IntelliTrace data, you must break execution of your application.

Break all

More options:

Open IntelliTrace settings

Learn more about IntelliTrace

```
'PhotoBrowser.exe' (Managed (v4.0.30319)):
Dispatched Thread ID: 3
'PhotoBrowser.exe' (Managed (v4.0.30319)):
'PhotoBrowser.exe' (Managed (v4.0.30319)):
'PhotoBrowser.exe' (Managed (v4.0.30319)):
'PhotoBrowser.exe' (Managed (v4.0.30319)):
'PhotoBrowser.exe' (Managed (v4.0.30319)):
```

Figure 13.2 The main page of the PhotoBrowser app, shown in snapped view alongside the desktop, which is in filled mode

that the user wants to keep going while they get real work done. In support of this, the app must surface only the information and functionality that makes sense when it's peripheral to the main activity for the user.

For our PhotoBrowser app's snapped state, we'll revisit the main page so that it shows categories rather than individual images. Similarly, the category browser page will show all images in a `ListView` rather than using the `FlipView`. Figure 13.2 shows the snapped view of the main page.

There are a number of different ways to do this. You could simply handle the state changes in code, and rearrange or otherwise show and hide items directly from the code-behind. The Visual Studio project templates have another approach, though, using a class called the `LayoutAwarePage`.

13.2 The LayoutAwarePage

The stock templates for any of the richer layout models come with a class called `LayoutAwarePage`. That class does a lot of great stuff for layout, as well as a lot of … interesting things for a generic MVVM implementation. Instead of using the `LayoutAwarePage` version of MVVM (of which I'm not a huge fan), you'll use MVVM Light. That means you'll also throw out all the layout goodness that comes with the page.

For this app, you'll create your own `LayoutAwarePage` but pull in only those things that will help you with view state management, and keep it simplified.

In the Common folder of the project, create a new class named `LayoutAwarePage` and add to it the following code.

Listing 13.1 A simplified `LayoutAwarePage` class

```
using System;
using System.Linq;
using Windows.UI.Core;
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace PhotoBrowser.Common
{
  public class LayoutAwarePage : Page
  {
    public LayoutAwarePage()
    {
      Window.Current.SizeChanged += this.WindowSizeChanged;

      Loaded += LayoutAwarePage_Loaded;
    }

    void LayoutAwarePage_Loaded(object sender, RoutedEventArgs e)
    {
      InvalidateVisualState();
    }

    private void WindowSizeChanged(object sender,
                                   WindowSizeChangedEventArgs e)
```

Catch size changes

```
    {
      InvalidateVisualState();
    }

    protected virtual string DetermineVisualState(
                      ApplicationViewState viewState)
    {
      return viewState.ToString();
    }

    public void InvalidateVisualState()
    {
      string visualState = DetermineVisualState(ApplicationView.Value);

      VisualStateManager.GoToState(this, visualState, false);
    }
  }
}
```

Check state and change if necessary

I kept this class pretty simple. The stock `LayoutAwarePage` does much more, such as keeping a list of layout-aware child controls that require notification. I encourage you to look at that class in the stock templates and either use it as is or modify it to suit your needs.

It may seem like a lot of simple functions in this code, but that's because I wanted to use the same function names and scope from the stock `LayoutAwarePage` class.

The `VisualStateManager.GoToState` method is responsible for setting the current state of this page to the specified visual state. The `false` parameter tells the function to not use any transition animations but instead to make the change happen immediately. You'll learn more about visual states shortly.

The first place you'll use this class is in MainPage.xaml. Change the opening and closing `Page` tags to `common:LayoutAwarePage`, and add the common namespace to the top. Also, name the grid that contains the `SemanticZoom` so you can refer to that from visual states later. The changes are all called out in the following listing.

Listing 13.2 MainPage.xaml page changes

```
<common:LayoutAwarePage
  x:Class="PhotoBrowser.MainPage"
  xmlns:common="using:PhotoBrowser.Common"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">
  ...
  <Grid Background="{StaticResource AppPageBackgroundBrush}">
    ...
    <Grid Grid.Row="1" x:Name="FullView">
      <SemanticZoom IsZoomOutButtonEnabled="True">
      ...
      </SemanticZoom>
    </Grid>
```

New base class

Base class namespace

Newly named grid

Existing markup

```
    <!-- snapped view -->

    <!-- visual states -->

  </Grid>
</common:LayoutAwarePage>
```

Snapped view will go here, inside the root grid

Visual states will go here, inside the root grid

Before this will compile, you'll need to add a `using` statement and also change the base class for the `MainPage` class declaration in the MainPage.xaml.cs code-behind file. The code-behind base class must always match the markup tag:

```
using PhotoBrowser.Common;
...
public sealed partial class MainPage : LayoutAwarePage
```

Finally, you want the UI elements in the snapped view to be synchronized with Semantic Zoom and everything else, in case the user bounces around between views. In support of that, you'll need to add a line of code to the `OnNavigatedTo` function in MainPage.xaml.cs. Here's the code.

Listing 13.3 Setting the `ItemsSource` in the code-behind

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  _vm.LoadPhotos();

  var cvs = Resources["PhotoSource"] as CollectionViewSource;

  SemanticZoomedOutView.ItemsSource =
      cvs.View.CollectionGroups;

  SnappedViewItems.ItemsSource =
      cvs.View.CollectionGroups;
}
```

New code to set ItemsSource for snapped view

Although the app as written won't yet compile, the `LayoutAwarePage` will make it really easy for you to implement visual states to handle the layout tasks.

Now you have the infrastructure in place to support the tracking of page view states. You also have placeholders in the XAML file to hold the snapped view and the visual states that will glue it all together.

13.3 The snapped view for the main page

The snapped view is the primary view you'll deal with throughout this chapter. You'll start simply, with the main page. You already started some of those changes when you changed the page to inherit from `LayoutAwarePage` instead of plain old `Page`.

In addition to the called-out changes in the MainPage.xaml markup, there are two placeholders in this file. The first is for the XAML you'll use for the snapped view. When you enter snapped view, you'll hide the full view grid and instead show the snapped view grid that contains your `ListView`.

A proper implementation of a snapped view almost always involves a completely separate control, or set of controls, from the full and filled views. The next listing has the XAML for the snapped view.

Listing 13.4 Snapped view XAML

```
<Grid Grid.Row="1" x:Name="SnappedView"
      Visibility="Collapsed">
  <ListView x:Name="SnappedViewItems"
            SelectionMode="None">
    <ListView.ItemTemplate>
      <DataTemplate>
        <Grid Width="280" Height="150"
              Margin="10,0,0,0">
          <Button Click="OnCategoryHeaderClick"
                  DataContext="{Binding Group}"
                  Foreground="Transparent"
                  BorderThickness="0"
                  Padding="0">
            <Grid>
              <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="25" />
              </Grid.RowDefinitions>

              <Image Grid.Row="0"
                     Grid.RowSpan="2"
                     Stretch="UniformToFill">
                <Image.Source>
                  <BitmapImage UriSource="{Binding Photos[0].ImageUri}"
                               DecodePixelWidth="280"/>
                </Image.Source>
              </Image>

              <Rectangle Grid.Row="1"
                         Fill="#FF0055AA"
                         Opacity="0.85"/>

              <TextBlock Grid.Row="1"
                         VerticalAlignment="Center"
                         Foreground="White"
                         Margin="5"
                         TextAlignment="Right"
                         TextWrapping="Wrap"
                         FontSize="12"
                         Text="{Binding Category}" />
            </Grid>
          </Button>
        </Grid>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</Grid>
```

Named grid

Button for click capability

Data context for event handler code

First image in group

Background rectangle

Category display

This markup has a grid with a `ListView` inside it instead of the `GridView` used for the full and filled views. If you followed along in the view controls chapter (chapter 10), this should be a pretty straightforward change. The `GridView` is optimized for horizontal scrolling, whereas the `ListView` is optimized for vertical scrolling—perfect for the snapped view.

Notice that the template for each item is just a big button with all the content inside it. This makes it easy to click an item and cause an action to happen. The `ListView` and `GridView` controls have an `ItemClick` event, but I wanted to be able to easily get the correct `DataContext` so you could reuse the click event handler from the `GridView`. As it turns out, drilling down from the `DataContext` on a grouped item from a `CollectionView` is somewhat tricky.

The app should now compile without errors. If you run the app, you still won't see any changes. You have all the controls in place, but you haven't put in anything to actually trigger the change. Right now you have everything in place except for the visual states that will show and hide the correct views.

13.4 Visual states for view management

Silverlight introduced (and WPF later adopted) the concept of visual states and the Visual State Manager (VSM). Prior to the adoption of VSM, everything was based on triggers. That is, if your mouse hovered over a button, a trigger would be fired to tell you to switch to the "mouse over" state. When the mouse moved off the button, another trigger would fire to tell you it was time to paint back in the normal state. When you started dealing with simultaneous states, the trigger implementation could be pretty hard to follow. It also wasn't really easy for visual designers to work with in tools like Expression Blend.

A visual state is implemented as an animation storyboard, but it's not really an animation. The storyboards have no real beginning or end; they're all immediate. In fact, the use of storyboards and animations here is just a convenient way to express a property change inside XAML, without involving any code.

Visual states are useful any place where you have a group of mutually exclusive states put together into a group. Within a single group, only one state may be active. If you have other states that can happen at the same time, such as focused and mouse over, you ensure the two states are in separate groups.

Initially, the Visual State Manager was used only for those small UI states like mouse movements, press, disabled, and so on. The team (and community) quickly realized that the concept was more useful than just that. In Windows 8, one place where visual states are being used is to manage the application view states.

To learn more about the Visual State Manager, you can visit: http://bit.ly/Win8XamlVSM. In addition, Silverlight references and examples are very close to what you'll use in Windows 8, with the primary difference being that no outer `VisualStateManager` tag is required in Windows 8 XAML apps.

The next listing shows the visual states in use on this page. I'll explain exactly what is happening right after the listing. Be sure to put this in the `CategoryBrowserPage` XAML in the spot reserved, inside the root grid.

Listing 13.5 Visual states

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ApplicationViewStates">
    <VisualState x:Name="FullScreenLandscape" />
    <VisualState x:Name="Filled" />
    <VisualState x:Name="FullScreenPortrait"/>
    <VisualState x:Name="Snapped">
      <Storyboard>

        <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="pageTitle"
            Storyboard.TargetProperty="Style">
          <DiscreteObjectKeyFrame
              KeyTime="0"
              Value="{StaticResource SnappedPageHeaderTextStyle}"/>
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="FullView"
            Storyboard.TargetProperty="Visibility">
          <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="SnappedView"
            Storyboard.TargetProperty="Visibility">
          <DiscreteObjectKeyFrame
              KeyTime="0"
              Value="Visible"/>
        </ObjectAnimationUsingKeyFrames>

      </Storyboard>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

State group name

Other states

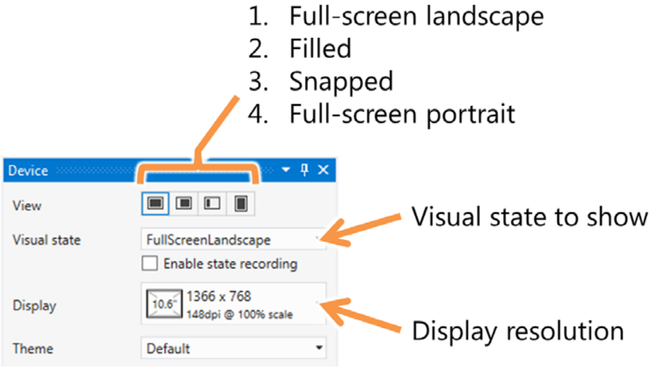
Snapped state

This `VisualStateGroup` contains four different states. Each state corresponds to one of the view states for the app (snapped, filled, and so on). Three of the states have no body. That means that for the objects referenced in the other states, the value for these empty states is the XAML as it's written at design time.

The fourth state has a `Storyboard` with several animations. Each of these changes happens simultaneously and instantly because you won't use any transition animations. Animations, and therefore visual states, can alter the value of any dependency property in any element on the page.

Here's what's happening in this state. When the state is activated, the following changes occur:

- The `Style` property of the `pageTitle` element is set to a static resource named `SnappedPageHeaderTextStyle`. This style makes the heading text smaller.



View
Visual state: FullScreenLandscape
☐ Enable state recording
Display: 10.6" 1366 x 768 148dpi @ 100% scale
Theme: Default

Figure 13.3 The device pane in Visual Studio. Because you're using visual states to control the view state, and because the state group is named `ApplicationViewStates`, the designer can work with your design and assist in previewing the different layouts.

- The `Visibility` property of the element named `FullView` (this is the grid containing the main `GridView` and `SemanticZoom` controls) is set to `Collapsed`.
- The `Visibility` property of the element named `SnappedView` is set to `Visible`.

Remember that, for all intents and purposes, these changes are all simultaneous.

Now, what happens if the app pops back into, say, the `FullScreenLandscape` state? The Visual State Manager handles working with the dependency properties to set them to their normal values automatically. You don't need to do that yourself. That's a huge time and markup saver.

Now, because you used the visual states here, and because you used the correct magic names for the states and the state group, the Device pane in Visual Studio has lit up with the ability to change the visual state right from the pane. Figure 13.3 shows what it looks like.

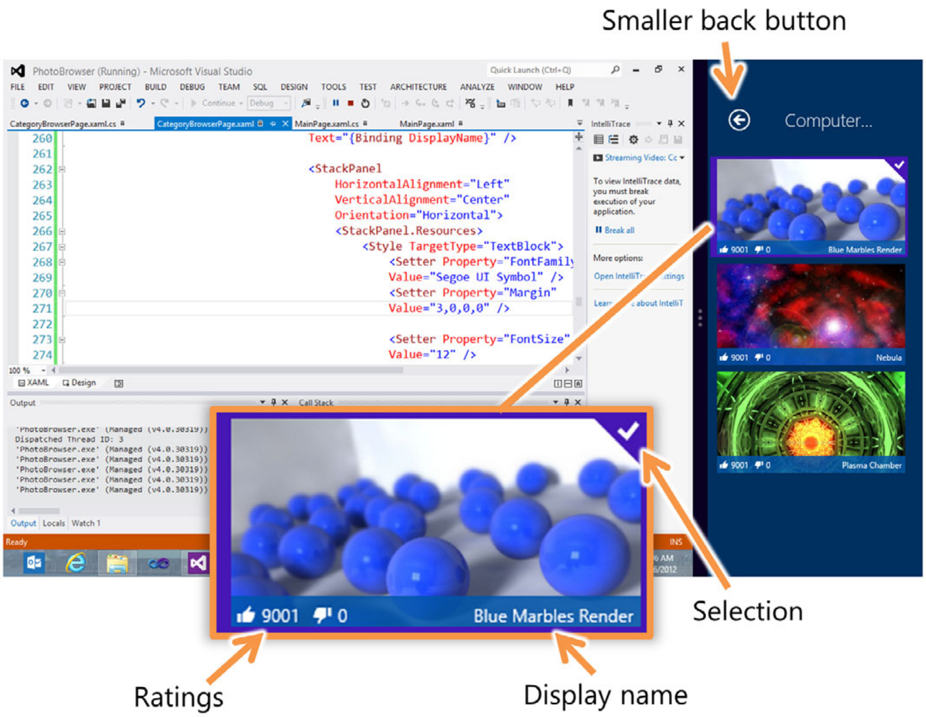
This is great for testing layouts under different states without running the app. In this case, because the layout is all data driven and because you don't have any design-time data, it's not going to show a whole lot. In a more complex app with more UI elements, or an app with design-time data, this pane becomes much more useful.

Now, run the app and snap it to an edge. The new layout will kick in, and you'll see a vertical list of image categories rather than the horizontal grid of categories and images. The Visual State Manager makes it really easy to wire up all the required changes directly from code, instead of having to reference a bunch of elements from code.

You've completed the main page. The next step will be to apply what you've learned here to the category browser page and then expand on that to handle the app bars.

13.5 Detail pages and app bars

The category browser page is somewhat more complex than the main page. This page has two different app bars and a layout that lends itself only to horizontal scrolling. The original version of this page (visible in chapter 12) has a number of commands on the app bar, a specialized navigation app bar at the top, and a sideways-scrolling `FlipView` for viewing the content. Clearly that's not all going to fit in the tiny space



Smaller back button
PhotoBrowser (Running) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL DESIGN TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Continue
Debug
CategoryBrowserPage.xaml.cs
CategoryBrowserPage.xaml
MainPage.xaml.cs
MainPage.xaml
IntelliTrace
100 %
XAML
Design
Output
Call Stack
Output
Locals
Watch 1
Ready
Computer...
9001
0
Blue Marbles Render
Nebula
Plasma Chamber
Selection
Ratings
Display name

```
                              Text="{Binding DisplayName}" />

                              <StackPanel
                                  HorizontalAlignment="Left"
                                  VerticalAlignment="Center"
                                  Orientation="Horizontal">
                                  <StackPanel.Resources>
                                      <Style TargetType="TextBlock">
                                          <Setter Property="FontFamily
                                          Value="Segoe UI Symbol" />
                                          <Setter Property="Margin"
                                          Value="3,0,0,0" />

                                          <Setter Property="FontSize"
                                          Value="12" />
```

```
'PhotoBrowser.exe' (Managed (v4.0.30319))
Dispatched Thread ID: 3
'PhotoBrowser.exe' (Managed (v4.0.30319))
'PhotoBrowser.exe' (Managed (v4.0.30319))
'PhotoBrowser.exe' (Managed (v4.0.30319))
'PhotoBrowser.exe' (Managed (v4.0.30319))
'PhotoBrowser.exe' (Managed (v4.0.30319))
```

Figure 13.4 The `CategoryBrowserPage` shown in snapped view. The control for this view is a `ListView`, not a `FlipView`, but all the binding works as before.

allotted to snapped views. What can you do to make this page useful and be sure it looks good even in this constrained view? Figure 13.4 is the look you're after (minus the app bars).

In this section you'll first work to create an appropriate control and layout for the display of the images. Once that's working, you'll tackle the app bar, all using visual states and the `LayoutAwarePage` created earlier.

13.5.1 Creating an appropriate presentation

The first task is to update the `CategoryBrowserPage` to use the `LayoutAwarePage` base class. You'll also use the same Visual State Manager approach as the main page. The next listing has the overall changes, including the updated page tags.

Listing 13.6 CategoryBrowserPage.xaml overall changes

```
<common:LayoutAwarePage
  x:Name="pageRoot"
  x:Class="PhotoBrowser.CategoryBrowserPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:common="using:PhotoBrowser.Common"
  xmlns:local="using:PhotoBrowser"
```

New base class

Common namespace

```
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
  ...
  <Grid ...>
    ...
    <Grid Grid.Row="1" x:Name="FullView">
      <FlipView ItemsSource="{Binding Category.Photos}" ... >
      ...
      </FlipView>
    </Grid>

    <!-- snapped view -->

    <!-- visual states -->

  </Grid>
</common:LayoutAwarePage>
```

Existing popup and row in here

New grid name

Snapped view placeholder

Visual states placeholder

This listing has the same types of changes as you had in `MainPage`. There's a new namespace declaration and a new base class, the primary content grid has been named `FullView`, and there are two placeholders to hold the snapped state control and the visual states.

In the code-behind, remember to set the base class of the `CategoryBrowserPage` to `LayoutAwarePage` just as you did with the main page.

The next listing has the XAML for the new snapped state visualization. This was a bit more challenging than the main page because there were more elements to consider, and the `FlipView` felt inappropriate in a snapped context. For those reasons, you'll go with another `ListView`.

Listing 13.7 CategoryBrowserPage.xaml snapped state control

```
<Grid Grid.Row="1" x:Name="SnappedView"
      Visibility="Collapsed">
  <ListView x:Name="SnappedViewItems"
            ItemsSource="{Binding Category.Photos}"
            SelectedItem="{Binding SelectedPhoto, Mode=TwoWay}">
    <ListView.ItemTemplate>
      <DataTemplate>
        <Grid Width="280" Height="150"
              Margin="10,0,0,0">
          <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="25" />
          </Grid.RowDefinitions>

          <Image Grid.Row="0"
                 Grid.RowSpan="2"
                 Stretch="UniformToFill">
            <Image.Source>
              <BitmapImage UriSource="{Binding ImageUri}"
                           DecodePixelWidth="280"/>
            </Image.Source>
          </Image>
```

SnappedView element

ListView

TwoWay binding

Image thumbnail

```
          <Grid Grid.Row="1">
            <Rectangle Fill="#FF0055AA"
                       Opacity="0.85"/>

            <TextBlock VerticalAlignment="Center"
                       Foreground="White"
                       Margin="5"
                       TextAlignment="Right"
                       TextWrapping="Wrap"
                       FontSize="12"
                       Text="{Binding DisplayName}" />

            <StackPanel HorizontalAlignment="Left"
                        VerticalAlignment="Center"
                        Orientation="Horizontal">
              <StackPanel.Resources>
                <Style TargetType="TextBlock">
                  <Setter Property="FontFamily"
                          Value="Segoe UI Symbol" />
                  <Setter Property="Margin"
                          Value="3,0,0,0" />
                  <Setter Property="FontSize"
                          Value="12" />
                </Style>
              </StackPanel.Resources>

              <StackPanel Orientation="Horizontal"
                          HorizontalAlignment="Left"
                          Margin="0,0,5,0">
                <TextBlock Text="&#xE19F;" />
                <TextBlock Text="{Binding LikesCount}" />
              </StackPanel>

              <StackPanel Orientation="Horizontal"
                          HorizontalAlignment="Left">
                <TextBlock Text="&#xE19E;"/>
                <TextBlock Text="{Binding DislikesCount}"/>
              </StackPanel>

            </StackPanel>
          </Grid>
        </Grid>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</Grid>
```

Rectangle overlay

Display name

Ratings layout

Likes rating

Dislikes rating

At first, this `ListView` appears very similar to the one on the main page. There are some important differences, however. This `ListView` supports item selection so that the `SelectedItem` property can be used to bind with the viewmodel. This `ListView` also has the Likes and Dislikes ratings rolled up and put into the same overlay bar as the image's `DisplayName`.

In this way, you're able to tailor the experience for the available space by using a completely different control. You don't have to change a thing in the viewmodel, or even in the code-behind, because this is completely a view-level change. It's nice to see MVVM working as promised.

You have the view in place, but you don't yet have it wired up with the appropriate visual states. This is easy to do and almost identical to the ones you used on MainPage.xaml. Here's the markup.

Listing 13.8 CategoryBrowserPage.xaml view states

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ApplicationViewStates">
    <VisualState x:Name="FullScreenLandscape" />
    <VisualState x:Name="Filled" />
    <VisualState x:Name="FullScreenPortrait"/>
    <VisualState x:Name="Snapped">
      <Storyboard>
        <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="backButton"
            Storyboard.TargetProperty="Style">
          <DiscreteObjectKeyFrame
              KeyTime="0"
              Value="{StaticResource SnappedBackButtonStyle}"/>     Smaller back button
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="pageTitle"
            Storyboard.TargetProperty="Style">
          <DiscreteObjectKeyFrame
              KeyTime="0"
              Value="{StaticResource SnappedPageHeaderTextStyle}"/>
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="FullView"               Full view
            Storyboard.TargetProperty="Visibility">
          <DiscreteObjectKeyFrame
              KeyTime="0"
              Value="Collapsed"/>
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames
            Storyboard.TargetName="SnappedView"            Snapped view
            Storyboard.TargetProperty="Visibility">
          <DiscreteObjectKeyFrame
              KeyTime="0"
              Value="Visible"/>
        </ObjectAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Now you can run the app, snap it to a side, and easily use both the main page and the category browser page.

There's one problem, however; the app bars aren't correct anymore. The top nav app bar is just about useless, and the bottom app bar no longer shows all the options. Plus, it's actually a bit hard to make the app bar appear. To make this view work, you'll need to adjust the app bars to fit the view.

TIP If you need to support more orientation logic than what's available in the view states, you can use the `DisplayProperties.CurrentOrientation` property or even one of the sensors that returns detailed orientation information.

13.5.2 Fixing up the app bar

Regardless of how well you scale the rest of the UI, one thing that almost always has to change is the app bar. A normal app bar in snapped state can fit approximately five normal-size buttons side by side, with labels hidden. That's not to say you want to just cram your current buttons in there, however. Because of the change in UI and the focus on things the casual user will want to do as opposed to the focused user, you'll need to rethink the options you supply to them.

You have choices when laying out the app bars. You can show the five buttons and hide the text labels, or you can show fewer buttons and keep the labels. Whenever possible, I opt for keeping the labels. That's what you'll do here, but that also means that you'll need to cut down on what you show on the app bar. I've also decided that the top app bar should be completely hidden in this view. Figure 13.5 shows the result you're after.

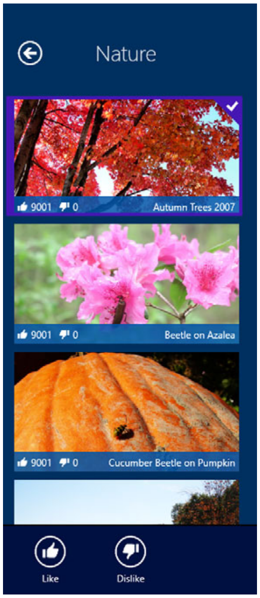


Figure 13.5 The bottom app bar with two commands. Note that there's no top app bar present.

For the top app bar, the change is simple: It needs to be named so you can refer to it in the visual state. The following listing has this minor change.

Listing 13.9 CategoryBrowserPage.xaml top app bar for snapped view

```
<Page.TopAppBar>
  <AppBar Background="#FF001040"
          Height="200"
          x:Name="FullTopAppBar">
    ...
  </AppBar>
</Page.TopAppBar>
```

New app bar name

You'll get to actually hiding it when you set up the visual states.

The next step is to update the bottom app bar by naming the buttons and separators so they can be referred to in the visual state. You'll also name the bottom app bar so you can set some properties on it. The only functions you'll want to enable in the snapped view are Like and Dislike. Pinning, deleting all photos, and rotating images will be available only in the wider views. The next listing shows the full bottom app bar with these updates.

Listing 13.10 CategoryBrowserPage.xaml bottom app bar

```
<Page.BottomAppBar>
  <AppBar Background="#FF001040"
          IsOpen="True" IsSticky="True"
          x:Name="FullBottomAppBar">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>

      <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Left"
                  Grid.Column="0">
        <Button x:Name="RotateSelectedPhoto"
                Style="{StaticResource RotateAppBarButtonStyle}"
                Command="{Binding RotateSelectedPhotoCommand}"/>

        <Rectangle x:Name="BottomAppBarSeparator1"
                   Margin="5,15,5,15"
                   Stroke="White"
                   StrokeThickness="2"
                   VerticalAlignment="Stretch"
                   Width="1"
                   Opacity="0.25"/>

        <Button x:Name="LikeSelectedPhoto"
                Style="{StaticResource LikeAppBarButtonStyle}"
                Command="{Binding LikeSelectedPhotoCommand}" />
        <Button x:Name="DislikeSelectedPhoto"
                Style="{StaticResource DislikeAppBarButtonStyle}"
                Click="OnDislikeClick" />
      </StackPanel>

      <StackPanel x:Name="BottomAppBarPageCommands"
                  Orientation="Horizontal"
                  HorizontalAlignment="Right"
                  Grid.Column="1">
        <Button Style="{StaticResource PinCategoryAppBarButtonStyle}"
                Command="{Binding PinCategoryCommand}" />
        <Rectangle Margin="5,15,5,15"
                   Stroke="White"
                   StrokeThickness="2"
                   VerticalAlignment="Stretch"
                   Width="1"
                   Opacity="0.25"/>
        <Button Style="{StaticResource ExterminateAppBarButtonStyle}"
                Command="{Binding DeleteAllPhotosCommand}" />
      </StackPanel>
    </Grid>
  </AppBar>
</Page.BottomAppBar>
```

App bar

Rotate button

Separator

Like button

Dislike button

Page-level commands on right

Other than naming a few elements, nothing has changed in the app bar markup since you first created the app bars.

The next listing has the additional animations required to be added into the storyboard in the snapped visual state. They can go anywhere inside there, as long as they're enclosed within the `Storyboard` tag in the snapped state.

Listing 13.11 CategoryBrowserPage.xaml visual states app bar entries

```
<ObjectAnimationUsingKeyFrames
    Storyboard.TargetName="RotateSelectedPhoto"
    Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>
</ObjectAnimationUsingKeyFrames>
```

Hide RotateSelectedPhoto

```
<ObjectAnimationUsingKeyFrames
    Storyboard.TargetName="BottomAppBarSeparator1"
    Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>
</ObjectAnimationUsingKeyFrames>
```

Hide separator

```
<ObjectAnimationUsingKeyFrames
    Storyboard.TargetName="BottomAppBarPageCommands"
    Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>
</ObjectAnimationUsingKeyFrames>
```

Hide page-level commands

```
<ObjectAnimationUsingKeyFrames
    Storyboard.TargetName="FullBottomAppBar"
    Storyboard.TargetProperty="IsSticky">
    <DiscreteObjectKeyFrame KeyTime="0" Value="True"/>
</ObjectAnimationUsingKeyFrames>
```

Make app bar sticky

```
<ObjectAnimationUsingKeyFrames
    Storyboard.TargetName="FullBottomAppBar"
    Storyboard.TargetProperty="IsOpen">
    <DiscreteObjectKeyFrame KeyTime="0" Value="True"/>
</ObjectAnimationUsingKeyFrames>
```

Set app bar to open by default

```
<ObjectAnimationUsingKeyFrames
    Storyboard.TargetName="FullTopAppBar"
    Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>
</ObjectAnimationUsingKeyFrames>
```

Hide top app bar

These visual states hide the separator and rotate button on the left and then hide the entire stack panel on the right side of the app bar. They also hide the top app bar by setting its `Visibility` to `Collapsed`. Finally, they set the bottom app bar to be open and sticky (not dismissed by clicking outside it) by default. The user can still dismiss the app bar manually with a right click, swipe, or keyboard shortcut.

You've made it so the app bar is always open. To make the app even more user friendly, you may want to increase the margin on the grid so the scrollbars are fully accessible even when the app bar is open.

Changing the app bar to work with the new layout awareness was actually quite simple. All you needed to do was make sure you could refer to the elements from within a visual state and then set up the visual state to show or hide them as appropriate. Because of the architecture of the app, you haven't had to change the viewmodels to support these new view states. Binding takes care of the magic for you.

App bar buttons and the stock LayoutAwarePage

The version of the `LayoutAwarePage` included in the stock templates has some interesting code that iterates through a collection of "layout-aware" controls when the layout changes. The app bar buttons, if using the default styles, qualify as layout-aware controls.

To get the buttons to tighten up and not use labels, wire up their `Loaded` event to the `StartLayoutUpdates` handler and their `Unloaded` event to the `StopLayoutUpdates` handler. This subscribes/unsubscribes them for layout updates, making it easy to let the snapped state layout happen automatically. Here's an example:

```
<Button Loaded="StartLayoutUpdates"
        Unloaded="StopLayoutUpdates"
        Style="{StaticResource PlayAppBarButtonStyle}" />
```

The trimmed-down version of the `LayoutAwarePage` in the examples here doesn't include this additional functionality, but it can be easily added by copying over from the stock `LayoutAwarePage` in a template project.

Regardless of whether you follow this approach, you'll still need to do much of the hard work described in this section, such as deciding which buttons should be displayed and ensuring they are appropriately shown or hidden.

13.6 Summary

Windows 8 Modern Style apps don't support overlapped windows. Instead, they support view states to allow more than one app on the screen at a time. This greatly simplifies touch-based interaction, while providing the flexibility to use two apps at once. If you've ever tried to perform real window management and docking using Windows 7 on a touch-based device, without benefit of keyboard or mouse, you know it can be a hassle. Windows 8 solves that by allowing apps to be full screen in portrait or landscape, snapped to the left or right, or filled to take up the remaining space when another app is snapped. Collectively, these are referred to as view states.

Supporting the different view states makes your app ready for the store. Supporting them in a meaningful way—providing useful functionality for each view—makes your app ready for your users.

Of the different states, the one that requires the most thought is the snapped state. This state restricts you to such a limited amount of real estate that you have to rethink your app's UI and interaction model and tailor it to fit that space. In particular, the app bar must be properly designed so that it both fits in the snapped state and makes sense in that state. Once you learn how to deal with the snapped state, you can make any number of changes to tailor the UI for the remaining states.

Your app must respond to this view state change by displaying not only an appropriately sized UI but also an appropriate experience. Apps that are snapped aren't the primary focus for the user and therefore should provide functionality that reflects this secondary focus. WinRT XAML provides events you can use to track the state changes and appropriately change the UI to fit.

When switching between states, the Visual State Manager can be a great way to organize all the changes for a specific state and to keep those changes in the markup. By keeping the states completely within the markup, your UI designer has complete freedom in how they design the states and the overall UI.

By properly supporting the view states, you can ensure your app looks good no matter how the user is using it or the device.

In the next chapter, we'll look at some of the contracts built into Windows 8, and you'll learn how to integrate the app with other apps in the system.

14

Contracts: playing nicely with others

This chapter covers

- Contracts
- Sharing with other apps
- Implementing search

Over the years, desktop apps have used a variety of mechanisms to integrate with each other. We've seen shared memory, shared memory mapped files, regular flat C-style DLLs, ActiveX, classic COM (Component Object Model) and other forms of COM automation like the much-hated DCOM (Distributed COM), DDE (Dynamic Data Exchange—remember that one?), socket communication, named pipes, and more. Each of these approaches had its good and bad points. Some, like DDE, were really brittle. Others, like COM, were complex to do well. Some like DCOM, no one ever really got working well. None of these mechanisms had any focus on the type of interaction; they were simply alternative ways to handle the app-to-app plumbing.

Windows 8 takes a different approach to app-to-app communication. Rather than focus on just the plumbing, the architects of Windows 8 took a step back and asked about the specific scenarios that should be supported. They then formalized the support for those into things called *contracts*.

Several of the more interesting contracts—Search, Share, and Settings, for example—are all exposed through the charms bar on the right (on a left-to-right machine) of the screen. Figure 14.1 shows the Windows charms bar.

I'll save the Settings contract for chapter 22. In this chapter, we'll look at a few of the other important contracts. First, we'll look at sharing files, links, and more with other apps. Then, we'll dive into what it takes to respond to sharing requests from other apps. From there, I'll show you how to let other apps search inside your application using the Search contract. This is a great new way to make sure your app is not just an island and to help increase the usefulness and number of uses of your app.

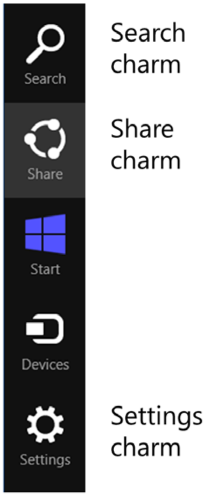

Search
Share
Start
Devices
Settings
Search charm
Share charm
Settings charm

Figure 14.1 The Search, Share, and Settings charms

14.1 Sharing

Most apps have data they can share with other apps. In some cases, it's a file created with the app. In others, it's a link to content online, a quote of the day, or a recommendation. With Windows 8 we're encouraged to think about the types of data that our apps can share. Each app that shares data makes the whole platform better and especially makes other apps more useful.

Another great benefit to sharing is that you don't need to know all the sharing endpoints at compile time. For a long time, on other platforms, each app had to build in sharing to major social networks and other endpoints. As those networks changed their APIs (Twitter and Facebook did regularly) or new networks were added, the apps had to be recompiled and redeployed with the new sharing feature. By standardizing the

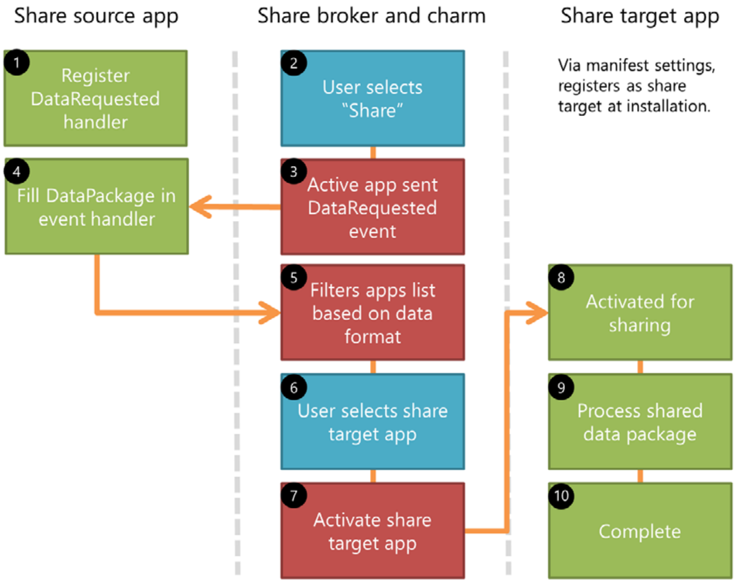

Share source app
Share broker and charm
Share target app
1 Register DataRequested handler
2 User selects "Share"
Via manifest settings, registers as share target at installation.
3 Active app sent DataRequested event
4 Fill DataPackage in event handler
5 Filters apps list based on data format
8 Activated for sharing
6 User selects share target app
9 Process shared data package
7 Activate share target app
10 Complete

Figure 14.2 Sharing illustrated. Diagram adapted from MSDN version at http://bit.ly/Win8AddingShare.

Share contract, you now let the apps that are designed specifically for those networks or endpoints handle keeping up to date with the APIs. This is a huge win for everyone.

Sharing is a system-level feature in Windows 8. Figure 14.2 shows how sharing is implemented using a broker between the source and target apps.

In this section, you'll learn how to be both a share source and a share target. You'll share photos data with other apps, and you'll also learn how to enable your app to be the recipient of data shared from other apps.

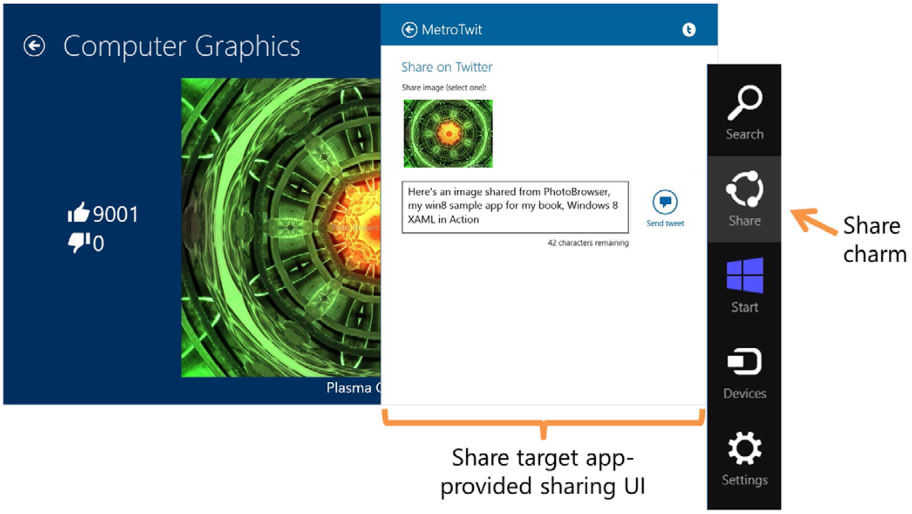
14.1.1 Sharing your data

Most of us learned to share around age five. For multiple reasons, however, we've forgotten that lesson when building apps. Sharing was never built into the platform or really encouraged, so I think we can all be forgiven for avoiding it. Until now, that is.

In Windows 8 it's incredibly easy to share data from your app. When your app shares data with other apps, it's considered a sharing source. A sharing source can provide data in a number of formats including plain text, URL, HTML, bitmap images, and files. Figure 14.3 shows the PhotoBrowser app sharing an image with another app.

Apps that create text can share that in its native format, for emailing or posting to a social network, for example. Similarly, apps that enable users to create images, like drawing apps, can provide those images to other apps to email, tweet, or post to a site. Music-creation apps can share their resulting MP3 files with apps that can share them on sites like SoundCloud. It's instant value for the user, value that increases as they add more apps to their system.

Sharing isn't added as a button in the app. Instead, it's invoked through the charm. Therefore, the app must listen to the charms bar to know when sharing is



Here's an image shared from PhotoBrowser, my win8 sample app for my book, Windows 8 XAML in Action

Share charm

Share target app-provided sharing UI

Figure 14.3 The PhotoBrowser app sharing an image with MetroTwit. The data format is an image stream and also includes a thumbnail version. Note that at the time of this writing, MetroTwit failed to post the image itself, so if that happens to you, it may still have a bug or limitation.

requested. For this app, you'll handle the sharing only on the `CategoryBrowserPage`. On that page, the user will be able to share the currently viewed image.

Rather than implement the functionality in the page code, it's going to be broken apart into two pieces. The first and most important part is the wire-up for the contract. That's added to the `CategoryBrowserViewModel`, as shown in the following listing.

Listing 14.1 `CategoryBrowserViewModel` share source code

```
...
using Windows.ApplicationModel.DataTransfer;
namespace PhotoBrowser.ViewModel
{
  public class CategoryBrowserViewModel : ViewModelBase
  {
    private DataTransferManager _dtm;

    public void RegisterForDataTransfer()
    {
      _dtm = DataTransferManager.GetForCurrentView();
      _dtm.DataRequested += OnShareDataRequested;
    }

    public void DeregisterForDataTransfer()
    {
      _dtm.DataRequested -= OnShareDataRequested;
    }

    ...

    private async void OnShareDataRequested(
            DataTransferManager sender,
            DataRequestedEventArgs args)
    {
      if (SelectedPhoto != null)
      {
        args.Request.Data.Properties.Title =
            SelectedPhoto.DisplayName;

        var deferral = args.Request.GetDeferral();

        try
        {
          args.Request.Data.Properties.Thumbnail =
            await ImageService.Current
              .GetThumbnailStreamFromPhotoAsync(SelectedPhoto);

          args.Request.Data.SetBitmap(
            await ImageService.Current
              .GetFileStreamFromPhotoAsync(SelectedPhoto));
        }
        catch (Exception ex)
        {
          Debug.WriteLine(ex.ToString());
        }
```

Share namespace

Data transfer manager

Data transfer wire-up

Called when transfer initiated

Friendly title for data

Deferral

Thumbnail

Photo

```
        finally
        {
          deferral.Complete();
        }
      }
      else
      {
        args.Request.FailWithDisplayText("No image selected.");
      }
    }
  }
}
```

Complete deferral

"No data" message

The code for sharing uses the deferral pattern. This is a requirement if there's asynchronous code as part of the sharing operation. Simply, the pattern involves getting a deferral and when the operation is done, marking the deferral as complete, much as you would in a transaction.

If, however, your sharing operation could take longer than 200 ms (you need to transcode a huge file, for example), you need to use the `SetDataProvider` method and a delegate to return the `DataPackage` to the target app. For more information on this approach, see http://bit.ly/Win8ShareSetDataProvider.

The heavy lifting of generating the data streams to share with the other app has all been done inside the `ImageService` class. This class needs a few changes as well, because all of your image manipulation so far has worked with URIs, not streams. Sending a URI to a local resource isn't going to work well with sharing, so you need to convert the bitmap to a more generic format. In support of this, the `ImageService` class needs to be updated to be able to supply the data from the image file. The next listing has the code you'll need.

Listing 14.2 Updated `ImageService` methods

```
public async Task<RandomAccessStreamReference>
             GetFileStreamFromPhotoAsync(Photo photo)
{
  var f = await StorageFile
      .GetFileFromApplicationUriAsync(photo.ImageUri);

  return RandomAccessStreamReference.CreateFromFile(f);
}
```

Stream from main image

```
public async Task<RandomAccessStreamReference>
             GetThumbnailStreamFromPhotoAsync(Photo photo)
{
  var f = await StorageFile
      .GetFileFromApplicationUriAsync(photo.ThumbnailUri);

  return RandomAccessStreamReference.CreateFromFile(f);
}
```

Stream from thumbnail image

To provide access to the `RandomAccessStreamReference` type, you'll need to add a `using` statement for the `Windows.Storage.Streams` namespace.

This code uses the helpful `GetFileFromApplicationUriAsync` method of the `StorageFile` object. The `StorageFile` class is the modern equivalent of a file in other APIs, but it now supports a greater variety of source locations and better security awareness. The function you're using understands app URIs and knows your app has permission to read from them. It's a safe API to use to quickly convert a URI into a stream of data.

> **IMPORTANT** It wasn't until this sample that I ran across a bug in the PhotoBrowser image loading. When creating the image URIs, I used `ms-appx:/` instead of `ms-appx:///`. Because of this, the code to load the images as a `StorageFile` was failing with an argument exception. In the `ImageService` class, add the additional slashes so the `uriRoot` variable is `"ms-appx:///Pictures/"` with three leading slashes, not just one.

In cases where having the user invoke the Share charm manually isn't the ideal solution—for example, a suggested share of a high score or an achievement—you can programmatically invoke the Share charm using this line of code:

```
Windows.ApplicationModel.DataTransfer.DataTransferManager.ShowShareUI();
```

Use this invocation method sparingly, and do avoid putting a share button in your app unless you're positive it makes sense for the user.

Finally, you need to add registration code in the CategoryBrowserPage.xaml.cs file. This is necessary because the data transfer manager can have only a single event handler listening at a time. If you try to wire up a second handler, you'll get an exception. The next listing has the updated methods to handle registration and deregistration.

Listing 14.3 Updated CategoryBrowserPage.xaml.cs navigation code

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  var vm = e.Parameter as CategoryBrowserViewModel;

  _vm = vm;

  DataContext = _vm;

  _vm.RegisterForDataTransfer();
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
  base.OnNavigatedFrom(e);

  _vm.DeregisterForDataTransfer();
}
```

Enable event handler

Disable and remove handler

With all of this in place, run the app. When you're on the category browser page, invoke the Share charm on the right. You'll see a list of apps on your PC that can accept the data that your app shares. This data is requested before the app list is

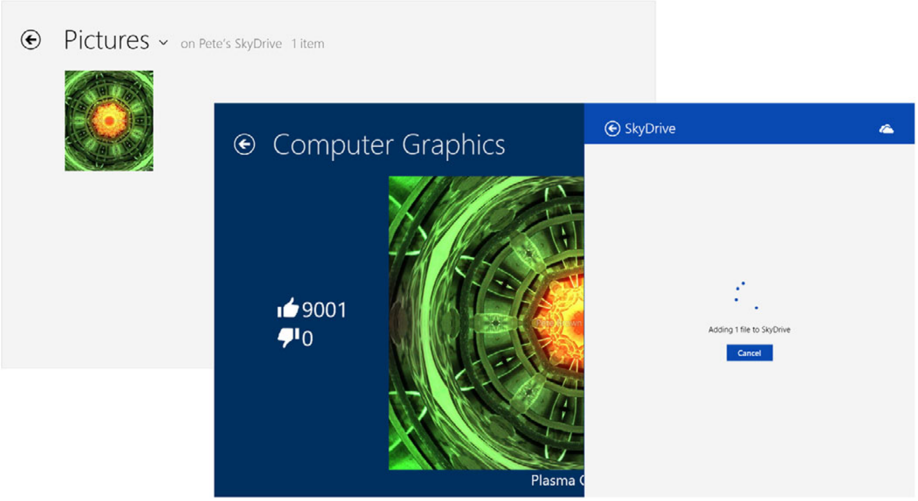


Figure 14.4 A successful share to SkyDrive, verified by seeing the file listed in my otherwise bare SkyDrive Pictures folder

shown, so that Windows knows how to filter the list so only appropriate ones are displayed. A successful share, for example to SkyDrive, is shown in figure 14.4.

> **TIP** You can use Windows + C to pop out the charms bar if you find the mouse interaction too fiddly. Or, more directly, you can use Windows + H to invoke the share charm.

Sharing data from your app adds instant value for your users. Now the data and artifacts they create are free. Sharing isn't just a one-way conversation. In many cases, it makes sense for an app to share with your app as the target.

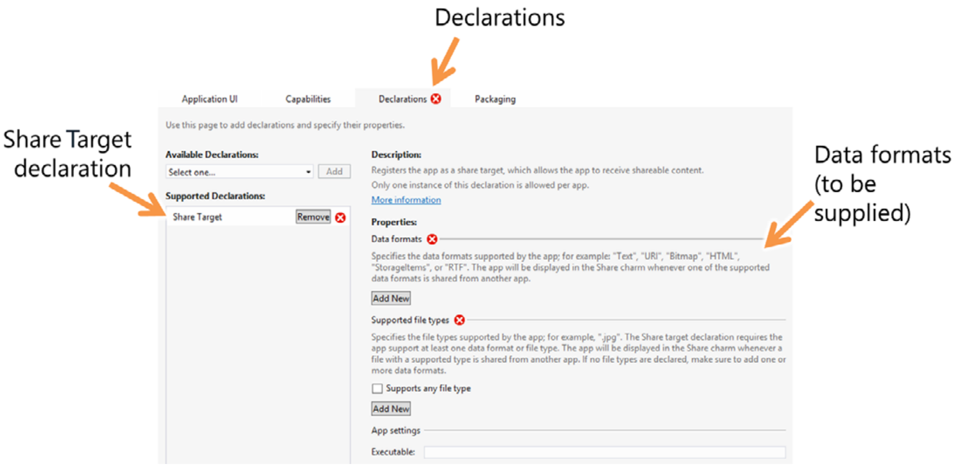
14.1.2 Letting others share with you

If your app catalogs data or connects to a service such as a social network, it's a good candidate for being a share target. For an app, being a share target means that other apps can send data to it in a number of formats and that your app does something meaningful with that data. In the previous examples, both MetroTwit and SkyDrive were shown as share targets. I'll leave it up to you to decide if posting to Twitter counts as "meaningful."

Our app isn't a great example of an app that could be a share target, because we don't write data to a service, store it locally, or otherwise do something useful. But, in the interest of stretching a demo app, we'll give it a try and at least receive data that we can display.

DECLARING THE APP AS A SHARE TARGET

The first thing you must do as an app developer is decide what types of data you'll support. Once you have that figured out, you must update the manifest to indicate that


Declarations

Share Target declaration

Data formats (to be supplied)

Application UI
Capabilities
Declarations
Packaging
Available Declarations:
Select one...
Add
Supported Declarations:
Share Target
Remove
Description:
More information
Properties:
Data formats
Add New
Supported file types
☐ Supports any file type
Add New
App settings
Executable:

Use this page to add declarations and specify their properties.

Registers the app as a share target, which allows the app to receive shareable content.

Only one instance of this declaration is allowed per app.

Specifies the data formats supported by the app; for example: "Text", "URI", "Bitmap", "HTML", "StorageItems", or "RTF". The app will be displayed in the Share charm whenever one of the supported data formats is shared from another app.

Specifies the file types supported by the app; for example, ".jpg". The Share target declaration requires the app support at least one data format or file type. The app will be displayed in the Share charm whenever a file with a supported type is shared from another app. If no file types are declared, make sure to add one or more data formats.

Figure 14.5 Adding the Share Target declaration to the appx manifest

the app is a share target for that type of data. Figure 14.5 shows this part of the manifest with the share target added but no data formats yet specified.

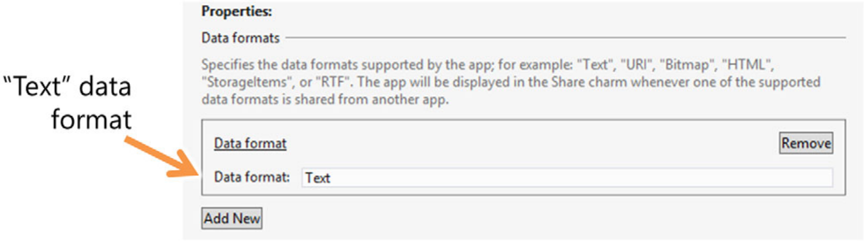
Once you add the Share Target declaration, you need to indicate how the data will be provided. For this app, you're going to use straight data, not file types. Additionally, the data format you accept will be plain text.

Under Data Formats in the manifest page, click Add New and enter *Text* into the Data Format field. Once you do that, all the red error X indicators will go away, and you'll know you've provided enough information to continue. Figure 14.6 shows the "Text" entry.

Apps can accept multiple formats. In those cases, simply add the additional formats as new entries. Apps can also accept files, filtered by file type, but you won't do that in this example.

CREATING THE SHARE TARGET PAGE

Each app that supports sharing must show a page to accept the data to be shared. The page could be as simple as a single button that accepts the share or as complex as an email message creation UI or more.


"Text" data format

Properties:
Data formats
Data format
Remove
Add New

Specifies the data formats supported by the app; for example: "Text", "URI", "Bitmap", "HTML", "StorageItems", or "RTF". The app will be displayed in the Share charm whenever one of the supported data formats is shared from another app.

Data format: Text

Figure 14.6 Selecting the Text type of acceptable data. How did I know to enter the magic string "Text"? It's in the helpful caption right above the field.

For the Photo Browser app, the page will be more along the lines of a debug page: You'll display the data that was shared to the app and the metadata that accompanies it. The UI is still a XAML page, but it will be displayed smaller than full screen.

Create a new XAML blank page named ShareTargetPage.xaml. The following listing has the markup for that file. Unlike the other pages, this page will have no title or navigation controls.

Listing 14.4 The ShareTargetPage.xaml markup

```
<Page
  x:Class="PhotoBrowser.ShareTargetPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PhotoBrowser"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

  <Grid Background="White">
    <Grid Margin="10">
      <Grid.Resources>
        <Style TargetType="TextBlock">
          <Setter Property="FontSize" Value="20" />
          <Setter Property="Margin" Value="10" />
          <Setter Property="Foreground" Value="Black" />
          <Setter Property="TextWrapping" Value="Wrap" />
        </Style>
      </Grid.Resources>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>

      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>

      <TextBlock Text="Source app" Grid.Row="0" Grid.Column="0"/>
      <TextBlock Text="Source app URI" Grid.Row="1" Grid.Column="0" />
      <TextBlock Text="Title" Grid.Row="2" Grid.Column="0" />
      <TextBlock Text="Description" Grid.Row="3" Grid.Column="0" />
      <TextBlock Text="Shared Text" Grid.Row="4" Grid.Column="0" />

      <TextBlock Text="{Binding ApplicationName}"
                 Grid.Row="0" Grid.Column="1" />
      <HyperlinkButton NavigateUri="{Binding ApplicationListingUri}"
                       Grid.Row="1" Grid.Column="1" >
```

White background

Text style

Field labels

Display fields

```
        <TextBlock Text="{Binding ApplicationListingUri}" />
      </HyperlinkButton>
      <TextBlock Text="{Binding Title}"
                 Grid.Row="2" Grid.Column="1" />
      <TextBlock Text="{Binding Description}"
                 Grid.Row="3" Grid.Column="1" />
      <TextBlock Text="{Binding Text}"
                 Grid.Row="4" Grid.Column="1" />

    </Grid>
  </Grid>
</Page>
```

Display fields

The standard for flyouts such as settings and share is to use black text on a white background. Adhering to that will help your app fit in visually. Windows will provide the title and navigation controls for the page, so I didn't need to include them in the listing.

You're going to have a viewmodel for this page. The viewmodel is really simple, but I want you to have an architecture you can build on for more complex sharing scenarios. The next listing has the viewmodel code. Add it to a new class named `ShareTargetViewModel` in the ViewModel folder.

Listing 14.5 The `ShareTargetViewModel` class

```
using GalaSoft.MvvmLight;
using System;
using System.Linq;
using Windows.ApplicationModel.DataTransfer;
using Windows.ApplicationModel.DataTransfer.ShareTarget;

namespace PhotoBrowser.ViewModel
{
  public class ShareTargetViewModel : ViewModelBase
  {
    private ShareOperation _operation;
    public ShareTargetViewModel(ShareOperation operation)
    {
      _operation = operation;

      ProcessOperation();
    }

    private async void ProcessOperation()
    {
      if (_operation.Data.AvailableFormats
           .Contains(StandardDataFormats.Text))
      {
        Text = await _operation.Data.GetTextAsync();

        ApplicationName =
             _operation.Data.Properties.ApplicationName;
        ApplicationListingUri =
             _operation.Data.Properties.ApplicationListingUri;
        Description = _operation.Data.Properties.Description;
        Title = _operation.Data.Properties.Title;
```

Check for text data

Get text data

Get metadata

```
      }
    }

    private string _text;
    public string Text
    {
      get { return _text; }
      private set { Set<string>(() => Text, ref _text, value); }
    }

    private string _applicationName;
    public string ApplicationName
    {
      get { return _applicationName; }
      private set { Set<string>(() => ApplicationName,
                    ref _applicationName, value); }
    }

    private Uri _applicationListingUri;
    public Uri ApplicationListingUri
    {
      get { return _applicationListingUri; }
      private set { Set<Uri>(() => ApplicationListingUri,
                    ref _applicationListingUri, value); }
    }

    private string _description;
    public string Description
    {
      get { return _description; }
      private set { Set<string>(() => Description,
                    ref _description, value); }
    }

    private string _title;
    public string Title
    {
      get { return _title; }
      private set { Set<string>(() => Title, ref _title, value); }
    }
  }
}
```

This viewmodel sets a bunch of bindable properties using information provided from the share operation passed into the constructor. You cache the operation in case you want to extend the class to perform more operations.

As usual, you'll have a little code-behind as well. There's just enough in the next listing to wire up with the viewmodel; you don't need to do anything else.

Listing 14.6 The ShareTargetPage.xaml code-behind

```
using PhotoBrowser.ViewModel;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
```

```
namespace PhotoBrowser
{
  public sealed partial class ShareTargetPage : Page
  {
    private ShareTargetViewModel _vm;
    public ShareTargetPage(ShareTargetViewModel vm)
    {
      this.InitializeComponent();

      _vm = vm;
      DataContext = _vm;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
    }
  }
}
```

Require viewmodel for construction

Set data context

The page and viewmodel are both set up. But that's not everything you need to do. The bit that glues this all together is the special form of activation in app.xaml.cs.

MODIFYING APP.XAML TO SUPPORT SHARE ACTIVATION

When the app is selected by the user as the target of a sharing operation, the `OnShareTargetActivated` method of the application object is executed. You need to override this method to provide the correct functionality to accept the share. For our app, the code builds the viewmodel, provides it to the `ShareTargetPage`, then launches that page. The following listing has everything you need to add to App.xaml.cs.

Listing 14.7 App.xaml.cs modifications to support sharing

```
protected override void OnShareTargetActivated(
                        ShareTargetActivatedEventArgs args)
{
  var vm = new ShareTargetViewModel(args.ShareOperation);

  var sharePage = new ShareTargetPage(vm);

  Window.Current.Content = sharePage;
  Window.Current.Activate();
  DispatcherHelper.Initialize();
}
```

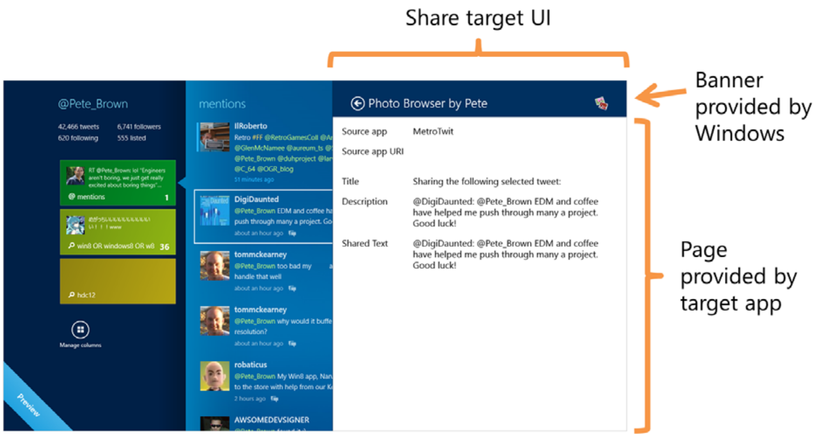
Create viewmodel

Create UI

Set window content

The code here is similar to what's in the `OnLaunched` method. This is simply another entry point into your app, so that makes complete sense. All the interesting bits related to the share operation are included in the args, specifically in the `ShareOperation` property. The `ShareOperation`, which is passed to the viewmodel, contains the actual data to be shared, as well as all the metadata about the share operation itself.

The app is now ready to run. If you want to debug it while using another app, use the same approach as you did for secondary tile activation: Set the app to debug without launching. Then, open up your favorite text-sharing-capable app, share something, and pick the PhotoBrowser as the target app. You'll see the UI shown in figure 14.7.


Share target UI
Banner provided by Windows
Page provided by target app
@Pete_Brown
mentions
Photo Browser by Pete

Source app MetroTwit
Source app URI

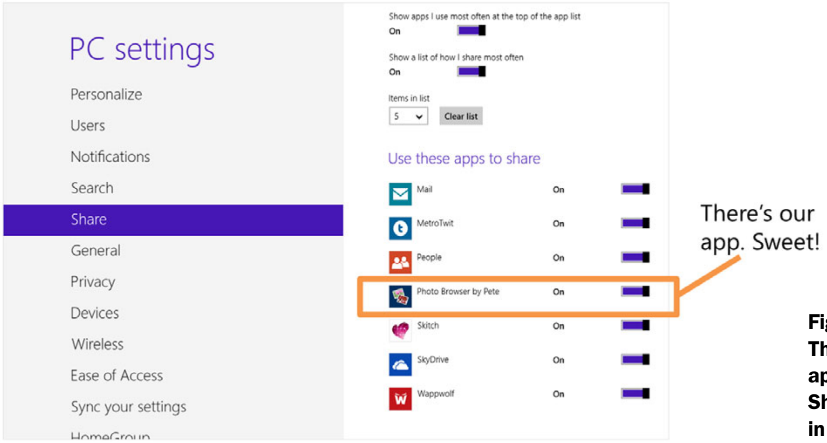
Title Sharing the following selected tweet:
Description @DigiDaunted: @Pete_Brown EDM and coffee have helped me push through many a project. Good luck!
Shared Text @DigiDaunted: @Pete_Brown EDM and coffee have helped me push through many a project. Good luck!

Figure 14.7 The PhotoBrowser app's Share Target UI shown with MetroTwit as the share source. Windows built and animated the flyout as well as its header.

Long-running share processing

Your app accepts simple data and doesn't do any processing of it. Let's say your app does something long-running with the data, like transcoding it or uploading the data to a service. In those cases, you'll want to use the `ReportStarted`, `ReportDataRetrieved`, `ReportError`, and `ReportSubmittedBackgroundTask` methods of the share operation. Those methods allow the app to continue processing the request without requiring that the user babysit the sharing page.

Because your app registered itself as a share target during install (based on the manifest settings), it now shows up in the Share settings in PC settings. This is the location where the user can turn on or off sharing for any individual app. Figure 14.8 shows the app in the list.


PC settings
Personalize
Users
Notifications
Search
Share
General
Privacy
Devices
Wireless
Ease of Access
Sync your settings
HomeGroup
Use these apps to share
Clear list
There's our app. Sweet!

Show apps I use most often at the top of the app list
On

Show a list of how I share most often
On

Items in list
5

Mail	On
MetroTwit	On
People	On
Photo Browser by Pete	On
Skitch	On
SkyDrive	On
Wappwolf	On

Figure 14.8 The PhotoBrowser app is shown in the Share targets list in PC settings.

The Share contract and the manifest settings together make it possible for your app to be a share target or a share source. You've not only created an interesting app, but you've also added value to many of the existing apps in the system by either feeding them data or serving as an endpoint for their app data.

Another approach to sharing app data is to be a search provider.

14.2 Letting others search your data

A great way to integrate with the rest of the system is to allow searching of the data within your app. The common search interface in Windows means that a user can concentrate on what they're searching for rather than what location to search in.

Search has a two primary use cases:

- The user searches for data while your app has focus. This frees you from adding search boxes to the app interface.
- The user searches for data from another app or the start page and selects your app to search within.

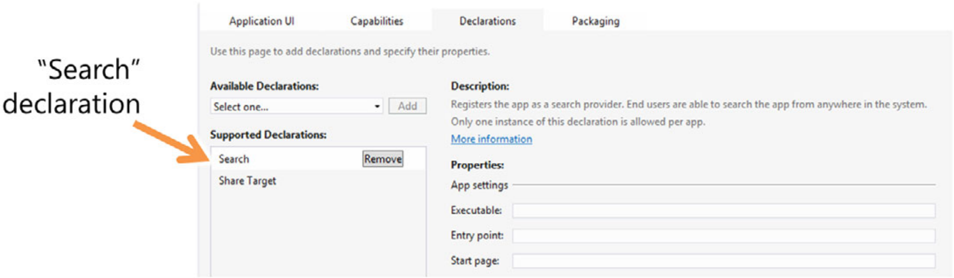
The approach for handling search, at least from an activation point of view, is different in each case.

In your app, you'll allow the user to search for images. This will require a search page as well as integration with the Search contract. In this section you'll build out a search interface that supports both the internal and external searches.

14.2.1 Declaring your intentions

As with share target integration, an app that is to be searchable must declare this in the appx manifest. As with the other declarations, there are a number of optional parameters for advanced scenarios. For your app, all you need to do is add the declaration and then save. Figure 14.9 shows the appx manifest with the correct declarations.

When adding search, you need to think about what you want to allow to be searched and how you will handle the results. You don't have multiple input fields; rather Windows uses the now-standard approach of a single field that will search all relevant data in the selected app. In this way, the user can easily switch between which apps they search without worrying about having different search criteria UI for each.



Use this page to add declarations and specify their properties.

Registers the app as a search provider. End users are able to search the app from anywhere in the system. Only one instance of this declaration is allowed per app.

Figure 14.9 Adding the Search declaration in the appx manifest

Beyond the simple search string, what the app does for search is entirely up to the app, because it owns the search results UI.

Before you implement the search itself, let's do the easy work and build out the results page. But, before doing that, take a moment to try search from the charms bar. Type in a simple bit of text, such as *File*. Now, switch between the search targets using the provided Windows 8 UI. I recommend trying at least the built-in Apps/Settings/Files but then also the Store app, Bing, and anything else you have that might be interesting. Notice how the Search UI is different for each of them. Some have filters; some have additional criteria. All declare the Search contract and provide a results page.

14.2.2 The results page and viewmodel

Your app doesn't have much data, so you can get away with just about anything on the results page. A well-designed results page should show the name of the app, as well as a subheading that indicates what search query the results are for. For large data sets, it should also support filtering of the results and should be sorted in a logical order—typically with the most relevant results first. You can read about these and other search design guidelines at http://bit.ly/Win8SearchGuidelines.

For this app, you'll create a simple search results page. Add a new blank XAML page named SearchResultsPage.xaml. The following listing shows the markup for this page.

Listing 14.8 SearchResultsPage.xaml

```
<common:LayoutAwarePage
  x:Class="PhotoBrowser.SearchResultsPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:PhotoBrowser"
  xmlns:common="using:PhotoBrowser.Common"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid Background="{StaticResource AppPageBackgroundBrush}">
    <Grid.RowDefinitions>
      <RowDefinition Height="140"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Grid Grid.Row="0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="116"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>

      <Button x:Name="backButton"
              IsEnabled="{Binding Frame.CanGoBack, ElementName=pageRoot}"
              Click="OnBackButtonClick"
              Style="{StaticResource BackButtonStyle}"/>
  <TextBlock x:Name="pageTitle" Text="Photo Browser Search"
             Grid.Column="1" IsHitTestVisible="false"
             Style="{StaticResource PageHeaderTextStyle}"/>
</Grid>

<GridView x:Name="SearchResultsView"
          Grid.Row="1"
          SelectionMode="None"
          IsItemClickEnabled="True"
          ItemClick="SearchResultsView_ItemClick"
          ItemsSource="{Binding SearchResults}">
  <GridView.ItemTemplate>
    <DataTemplate>
      <Grid Width="280" Height="150"
            Margin="10,0,0,0">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="25" />
          </Grid.RowDefinitions>
          <Image Grid.Row="0" Grid.RowSpan="2"
                 Stretch="UniformToFill">
            <Image.Source>
              <BitmapImage UriSource="{Binding ImageUri}"
                           DecodePixelWidth="280"/>
            </Image.Source>
          </Image>

          <Grid Grid.Row="1">
            <Rectangle Fill="#FF0055AA" Opacity="0.85"/>
            <TextBlock VerticalAlignment="Center"
                       Foreground="White"
                       Margin="5"
                       TextAlignment="Right"
                       TextWrapping="Wrap"
                       FontSize="12"
                       Text="{Binding DisplayName}" />

            <StackPanel HorizontalAlignment="Left"
                        VerticalAlignment="Center"
                        Orientation="Horizontal">
              <StackPanel.Resources>
                <Style TargetType="TextBlock">
                  <Setter Property="FontFamily"
                          Value="Segoe UI Symbol" />
                  <Setter Property="Margin"
                          Value="3,0,0,0" />
                  <Setter Property="FontSize"
                          Value="12" />
                </Style>
              </StackPanel.Resources>
              <StackPanel Orientation="Horizontal"
                          HorizontalAlignment="Left"
                          Margin="0,0,5,0">
                <TextBlock Text="&#xE19F;" />
                <TextBlock Text="{Binding LikesCount}" />
              </StackPanel>
```

Search results grid view

Item clicking enabled

Item click event handler

Same template as snapped view

```
                  <StackPanel Orientation="Horizontal"
                              HorizontalAlignment="Left">
                    <TextBlock Text="&#xE19E;"/>
                    <TextBlock Text="{Binding DislikesCount}"/>
                  </StackPanel>
                </StackPanel>
              </Grid>
            </Grid>
          </Grid>
        </DataTemplate>
      </GridView.ItemTemplate>
    </GridView>
  </Grid>
</common:LayoutAwarePage>
```

The search results page uses the same items template that you used for the snapped view of data in the previous chapter. But instead of a `ListView`, you use a `GridView` so the data will naturally span the screen. Unlike the other grids and lists, this one has item clicking enabled and wired up to an event handler in the code-behind. You could have used a button as before, but I wanted you to see how the item click functionality works in a `GridView`.

When the user clicks an item, the code-behind will make calls to the `SearchViewModel` and navigate to the category browser page with the item preselected and visible.

The next listing includes the new `SearchViewModel`. Create this as a class in the ViewModel folder alongside the others.

Listing 14.9 `SearchViewModel`

```
using GalaSoft.MvvmLight;
using PhotoBrowser.Model;
using PhotoBrowser.Services;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.UI.Xaml.Data;

namespace PhotoBrowser.ViewModel
{
  public class SearchViewModel : ViewModelBase
  {
    private IEnumerable<Photo> _searchResults;
    public IEnumerable<Photo> SearchResults
    {
      get { return _searchResults; }
      private set
      {
        Set<IEnumerable<Photo>>(() => SearchResults,
                     ref _searchResults, value);
      }
    }
```

Search results IEnumerable

```
    private string _queryText;
    public string QueryText
    {
      get { return _queryText; }
      set
      {
        Set<string>(() => QueryText, ref _queryText, value);
                UpdateSearchResults();
      }
    }
```

User-entered search string

```
    private void UpdateSearchResults()
    {
      var photos = ImageService.Current.GetPhotos();

      if (string.IsNullOrEmpty(QueryText))
        SearchResults = photos;
      else
      {
        var q = QueryText.ToLower();
```

Return all photos on empty search (first click)

```
        var results = from Photo p in photos
                      where (p.DisplayName.ToLower().Contains(q) ||
                             p.Category.ToLower().Contains(q))
                      select p;
        SearchResults = results;
      }
    }
```

Filter results

Build viewmodel

```
    public CategoryBrowserViewModel ConstructCategoryBrowserViewModel(
                                                  Photo photo)
    {
      var vm = new CategoryBrowserViewModel();

      foreach (PhotoCategory c in ImageService.Current.GetCategories())
        vm.AllCategories.Add(c);

      vm.Category = vm.AllCategories
                .FirstOrDefault<PhotoCategory>(
                    c => c.Category == photo.Category);

      vm.SelectedPhoto = photo;

      return vm;
    }
  }
}
```

The `SearchViewModel` is where most of the search action happens. Upon receiving a query string (a search word) from the `App` object, this class builds an `IEnumerable` of photos that match the result.

Most of the time when I bind to a collection, I use an observable collection and add/remove items individually. In the case of search, where the results will be different for each search issue, it makes more sense to simply rebuild the collection. For those reasons, I exposed the property as an `IEnumerable` rather than an `ObservableCollection` and provided a getter and setter for it. The collection is rebuilt in the `UpdateSearchResults` method.

The `ConstructCategoryBrowserViewModel` method is used when clicking an item in the results list. If you recall, the category browser page requires a populated viewmodel. This is the method that populates it, ensuring the correct photo and category are both selected.

The passing of the populated viewmodel happens in the `SearchResultsPage` code-behind, as shown here.

Listing 14.10 SearchResultsPage.xaml.cs code-behind

```
using PhotoBrowser.Common;
using PhotoBrowser.Model;
using PhotoBrowser.ViewModel;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace PhotoBrowser
{
  public sealed partial class SearchResultsPage : LayoutAwarePage
  {
    private SearchViewModel _vm;

    public SearchResultsPage()
    {
      this.InitializeComponent();

      _vm = new SearchViewModel();
      DataContext = _vm;
    }
```

Page creates viewmodel

```
    private void OnBackButtonClick(object sender,
                      Windows.UI.Xaml.RoutedEventArgs e)
    {
      Frame.GoBack();
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
      string queryText = e.Parameter as string;
      _vm.QueryText = queryText;
    }
```

Handle search parameter

```
    private void SearchResultsView_ItemClick(object sender,
                      ItemClickEventArgs e)
    {
      var photo = e.ClickedItem as Photo;
      var vm = _vm.ConstructCategoryBrowserViewModel(photo);
      Frame.Navigate(typeof(CategoryBrowserPage), vm);
    }
  }
}
```

Show category browser page on item click

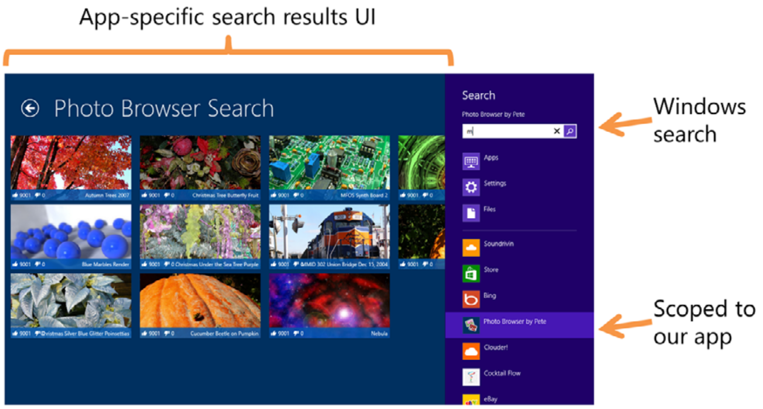


Figure 14.10 The Search UI invoked from within the app

As is usual for my examples, the code-behind has a little bit of glue code in place but not much logic. This code-behind handles creating the viewmodel and then dealing with navigation. When the page is navigated to, you pull the search string out of the parameters passed from the `App` object. When an item in the `GridView` is clicked, you build the viewmodel and navigate away to the category browser page.

If you run the app right now, you won't see anything. That's because I've left out the key part: the launching and parameter passing in app.xaml.cs.

14.2.3 Responding to in-app search requests

The simplest example of search is the in-app search request. This happens when the user already has your app open and focused and then chooses to search within it. In this case, you know the app is already loaded and activated. All you need to do is get the search string and navigate to the search results page, passing along the search string.

Figure 14.10 shows how the search interface looks in this app.

The code to glue together everything you've done so far is in the next listing, with an explanation after the listing.

Listing 14.11 App.xaml.cs in-app searching support

```
protected override void OnWindowCreated(WindowCreatedEventArgs args)
{
  base.OnWindowCreated(args);

  var pane = Windows.ApplicationModel.Search
                 .SearchPane.GetForCurrentView();

  pane.QuerySubmitted += OnSearchQuerySubmitted;
}

void OnSearchQuerySubmitted(
   Windows.ApplicationModel.Search.SearchPane sender,
   Windows.ApplicationModel.Search.SearchPaneQuerySubmittedEventArgs args)
{
```

Window creation override

Get search panel

```
  var previousContent = Window.Current.Content;
  var frame = previousContent as Frame;

  frame.Navigate(typeof(SearchResultsPage), args.QueryText);

  Window.Current.Content = frame;
  Window.Current.Activate();
}
```

Get previous content and frame

Navigate to search results page

Activate search page

Upon the creation of the top-level app window, the app registers for search handling. This wire-up happens once in the `OnWindowCreated` override. The event, `QuerySubmitted`, is the most important event for search. This is what gets fired whenever the user clicks the little search icon beside the search entry field.

The handler for this event takes the search text and passes it to the `SearchResultsPage` as part of an otherwise normal navigation action. As elsewhere in app.xaml.cs, you ensure the page has been activated before exiting the method.

To test, invoke the Search charm while the app is open and focused. The shortcut key for search is Windows + Q. As with the other examples, if you want to debug the app in use, you'll need to start debugging without launching the app.

Now that you have search working inside the app, it will take only a minor effort to integrate with the larger search system as a whole.

14.2.4 Responding to external search requests

The actual searching from external sources is identical to the search run within the app. But in addition to the `OnSearchQuerySubmitted` handler used for in-app searching, there's an activation step when search is external because this may be the first time the app was launched. Windows will first fire off `OnSearchActivated` and then the search query event and its event handler. For each subsequent search within the session, only the event will be fired.

The following listing has the activation method specific to search. There's some duplication of code here between the search query handler and this activated method. You can consolidate the common code into a single method if you wish.

Listing 14.12 App.xaml.cs support for external search

```
protected override void OnSearchActivated(
    Windows.ApplicationModel.Activation.SearchActivatedEventArgs args)
{
  var previousContent = Window.Current.Content;
  var frame = previousContent as Frame;

  if (frame == null)
    frame = new Frame();

  frame.Navigate(typeof(SearchResultsPage), args.QueryText);

  Window.Current.Content = frame;
  Window.Current.Activate();
}
```

Get window content

Get navigation frame

Create frame if new activation

Navigate to search page

Much of the code in this method is similar to the normal launch code in app.xaml, except this method is called on an external search activation. Use this to perform the same types of startup tasks, even showing the extended splash screen if necessary. But do the bare minimum amount of work required to quickly display the search results.

As before, to test, invoke the Search charm, but do so from any other app or from the desktop. As with the other external activation examples, if you want to debug the app in use, you'll need to start debugging without launching the app.

Most apps have data that's interesting to the user. By enabling search of that data, the app integrates better with the rest of the system and is more likely to be used, because the user has found it to be the source of the data they're looking for. The standard in-app and external search mechanisms provided in Windows 8 make it simple to implement search in your own app. For most apps, the addition of search will be an incremental effort beyond everything else the app already supports.

Project item templates for search and share

If you use the default template page styles (and their implementation of state and MVVM) in your app, you may want to consider using the Search Contract and Share Contract templates available in Visual Studio. To use them, use File > Add New Item, and pick the contract you want to add.

If you're not using the default templates (as is the case in my examples), using those templates is far more trouble than it's worth. The contracts are very easy to implement if you're following a solid MVVM pattern approach, so templates aren't required, in my opinion.

14.3 Summary

Apps are not islands, isolated from other apps in the system. Neither is the opposite true: Apps don't use the free-for-all integration mechanisms we've used for desktop applications over the years. Instead, Windows Store apps use standardized mechanisms for sharing information with other apps. These mechanisms are collectively referred to as contracts. Contracts are bigger than just app integration, but app integration is one of the largest use cases by far.

Through contracts, third-party apps that you've never seen, that perhaps were written long after your app was written, can share data and files with your app. Similarly, your app can share with them. This is really important when you think about all the new cloud services and social networking sites that spin up during an operating system's lifecycle. Now you can use all of them without needing to know about them in advance.

Another thing contracts enable is searching of data. When a user is looking for something in Windows 8, they'll use the Search charm. By integrating with the Search charm, you open up your app to the world; you're no longer an island of data, searchable only with the app's internal tools.

There are other contracts as well. Some, like the Print Task contract and the Settings contract, are less about app-to-app integration and more about simply providing standard ways to perform functions. Others, such as app-to-app picking, are very useful and follow similar patterns to what was discussed here. For more information on the full set of contracts, please see http://bit.ly/Win8Contracts. In the next chapter, before we look at additional contracts in subsequent chapters, we'll look at working with files.

15

Working with files

This chapter covers

- Listing files and folders
- Programmatically accessing files
- URI standards for file access
- Using and creating file pickers

Quick! Look at all the applications you currently have open on your computer. How many of them have files open? Those files may be web pages from a remote location, a custom database of email items, a document, a music file, maybe more. Even games need to access data files both for saved games as well as for just general game data.

Most apps need to work with files either as native data for the app or as something they consume and provide value by working with, like music files. Unlike web apps and plug-ins, native apps can be given more access to the filesystem, to load a number of different types of data files. As you'd expect, WinRT XAML apps can use a number of different approaches to working with these files.

The first approach we'll explore in this chapter is also the most involved: working with the `StorageFile` and `StorageFolder` classes. We'll use these classes, as well as a few other helper classes, to list files and to create new files. Along the way, we'll take a look at how to use multiple data templates to show files and folders differently in a single `ListView`.

Once we've worked out how to programmatically access files, we'll take a brief look at the URI syntax for the different file locations. Windows 8 adds a number of different URI schemes that you can use from markup as well as from code.

Desktop developers are used to having generally unfettered access to files. But we know that isn't safe for casually downloaded and installed apps. Instead, we need to have a mechanism by which we can specify what locations the app expects to access, and then allow the user to make the decision as to whether they trust the app enough to download and install it with those requested permissions granted. We'll cover those capabilities and declarations in this chapter.

Finally, one of the safest, and richest, approaches for working with files is to use the file picker. By using existing file pickers, you enable your app to use other apps as the source of the file data. Those files could reside anywhere, as long as that app has access to them. Your app can also be a file picker, so we'll look at that as well.

Let's start our exploration of file management by looking at how to work with files from code, without user intervention.

15.1 Loading files programmatically

In the past, if you wanted to open a file, you had to be able to resolve the name to a path and filename that APIs could understand. Networking of any sort was a completely separate operation. The WinRT API was designed to provide equal access to files located via filename and files located via URI. Even local files can often be referenced via a specific URI scheme.

In several of the previous chapters you performed basic file access using URIs and files inside the same app. For most of those, you used the `ms-appx` URI scheme. But there are lots of other ways to load data, programmatically as well as using other URI formats.

In this section, we'll look at how to list files and create them programmatically. First, you'll set up a new project using a basic viewmodel to give you a place to bind to. Then you'll add in the code to list files in three different well-known locations on the drive. But, to access those locations, you'll need to set the appropriate capabilities and declarations in the appx manifest, so you'll do a little of that here as well.

Once you have access, you'll dive into the `StorageFile` and `StorageFolder` classes. When working with files from code, these will be the two classes you use more than any others. You'll use the `StorageFolder` class to list all the files in a folder using a couple of different approaches. Then, you'll use it to create a new file, a `StorageFile`, which you'll then write to using the `FileIO` class.

15.1.1 New demonstration project

For this chapter, you'll create a new demonstration project. This will be a simple, from-scratch, lightweight MVVM implementation, without any third-party toolkits. It won't use commanding, because that would only clutter the example. But it will use a viewmodel and binding.

Create a new app named DemoApp. The template to use is the Blank App (Windows Store) template. Once it's open, create a folder named ViewModel and in that place a new class named `MainViewModel`. The code for this class is shown in the first listing.

Listing 15.1 `MainViewModel` code

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Storage;
using Windows.Storage.Search;

namespace DemoApp.ViewModel
{
  public class MainViewModel: INotifyPropertyChanged
  {
    private StorageFolder _currentFolder;
    public StorageFolder CurrentFolder
    {
      get { return _currentFolder; }
      set
      {
        _currentFolder = value;
        NotifyPropertyChanged("CurrentFolder");
        UpdateChildren();
      }
    }

    private ObservableCollection<IStorageItem> _childItems =
            new ObservableCollection<IStorageItem>();
    public ObservableCollection<IStorageItem> ChildItems
    {
      get { return _childItems; }
    }

    private async void UpdateChildren() { }
    public void LoadDocumentsLibrary() { }
    public void LoadMusicLibrary() { }
    public void LoadRemovableDevices() { }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void NotifyPropertyChanged(string propertyName)
    {
      if (PropertyChanged != null)
        PropertyChanged(this,
          new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

Supports change notification

Current parent folder

Load child items

Child items of CurrentFolder

To be provided

INotifyPropertyChanged implementation

The viewmodel follows a typical pattern: a list of data items and a current or selected item, plus methods for populating the data. In this case, the current item is a `StorageFolder` instance, and the child items list is a collection of objects that implement `IStorageItem`. You're going with the interface here to support both `StorageFolder` and `StorageFile` instances in the same collection.

Several of the methods will be implemented later in this chapter. The `UpdateChildren` method, because it is marked as async, will cause a compiler warning until you provide the asynchronous code.

Next, change the MainPage.xaml markup to match that in the following listing.

Listing 15.2 MainPage.xaml markup

```
<Page
  x:Class="DemoApp.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:DemoApp"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Page.Resources>
    <Style TargetType="Button">
      <Setter Property="Width" Value="200" />
      <Setter Property="Height" Value="50" />
      <Setter Property="Margin" Value="10" />
    </Style>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <Grid Grid.Column="0" Margin="30">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>

      <TextBlock Text="{Binding CurrentFolder.Name}"
                 FontSize="42" Grid.Row="0" />

      <!-- ListView goes here -->

    </Grid>

    <Grid Grid.Column="1" Margin="30">
      <StackPanel>
        <Button Content="Get Documents Library"
                Click="GetDocumentLibraryClick"/>
```

Displays current folder name

ListView to be provided

Buttons to load data

```
        <Button Content="Get Music Library"
                Click="GetMusicLibraryClick"/>
        <Button Content="Get Removable Devices"
                Click="GetRemovableDevicesClick"/>

   <!-- Additional buttons will go here -->

      </StackPanel>
    </Grid>
  </Grid>
</Page>
```

Buttons to load data

Additional buttons and input will go here

The page markup includes a `TextBlock` that displays the name of the currently selected folder, a placeholder for a `ListView` to display the items in the folder, and then a set of onscreen buttons to load the data.

Finally, update the MainPage.xaml.cs code-behind to wire everything together. Here's the new code.

Listing 15.3 MainPage.xaml.cs code-behind

```
using System;
using DemoApp.ViewModel;
using Windows.Storage;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace DemoApp
{
  public sealed partial class MainPage : Page
  {
    private MainViewModel _vm = new MainViewModel();

    public MainPage()
    {
      this.InitializeComponent();

      DataContext = _vm;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    { }

    private void GetDocumentLibraryClick(object sender, RoutedEventArgs e)
    {
      _vm.LoadDocumentsLibrary();
    }

    private void GetMusicLibraryClick(
                      object sender, RoutedEventArgs e)
    {
      _vm.LoadMusicLibrary();
    }
```

Create viewmodel

Set data context

Load documents library items

Load music library items

```
    private void GetRemovableDevicesClick(
                      object sender, RoutedEventArgs e)
    {
      _vm.LoadRemovableDevices();
    }
  }
}
```

Load removable device items

The code-behind creates an instance of the viewmodel and sets it as the data context for the entire page. There are also three event handlers on the page to handle the button clicks. Each calls a method on the viewmodel to load the data.

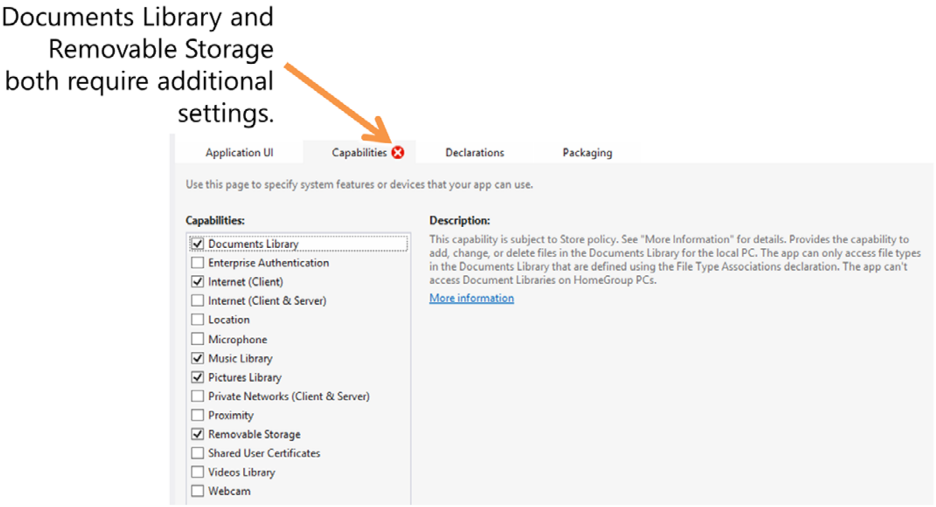
Each of these calls requires settings in the manifest so that the user can make informed decisions about what capabilities the app requires.

15.1.2 File access permissions

Windows Store apps can't access any random file on the local machine. Instead, they can access files only in certain known locations. Furthermore, this access is restricted based on the type of account under which the app was posted to the Windows Store. For example, only business accounts can create apps that have programmatic access to the documents library (individual accounts can still create apps that use the documents library via a file picker).

Access to the Music, Pictures, and Videos libraries is easily set using the Capabilities tab in the appx manifest, as shown in figure 15.1.

Programmatic access to the Documents Library and Removable Storage capabilities both require additional file association settings. As previously mentioned, apps that request these settings can only be placed into the Windows Store through a busi-



Documents Library and Removable Storage both require additional settings.

Use this page to specify system features or devices that your app can use.

Capabilities:

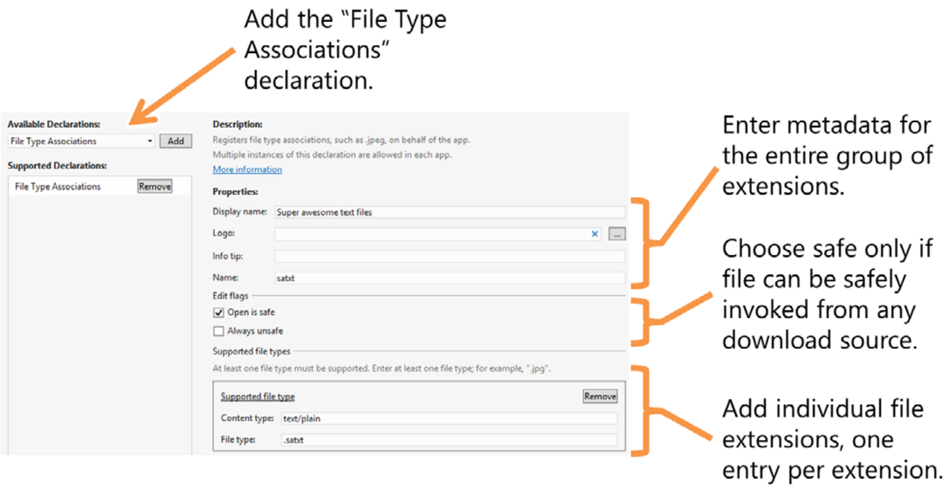
☑ Documents Library
☐ Enterprise Authentication
☑ Internet (Client)
☐ Internet (Client & Server)
☐ Location
☐ Microphone
☑ Music Library
☑ Pictures Library
☐ Private Networks (Client & Server)
☐ Proximity
☑ Removable Storage
☐ Shared User Certificates
☐ Videos Library
☐ Webcam

Description:

This capability is subject to Store policy. See "More Information" for details. Provides the capability to add, change, or delete files in the Documents Library for the local PC. The app can only access file types in the Documents Library that are defined using the File Type Associations declaration. The app can't access Document Libraries on HomeGroup PCs.

More information

Figure 15.1 Declaring capabilities for folder access. Documents Library and Removable Storage capabilities require additional settings.



Add the "File Type Associations" declaration.

Enter metadata for the entire group of extensions.

Choose safe only if file can be safely invoked from any download source.

Add individual file extensions, one entry per extension.

Available Declarations: File Type Associations

Supported Declarations: File Type Associations

Description:

Registers file type associations, such as .jpeg, on behalf of the app.

Multiple instances of this declaration are allowed in each app.

More information

Properties:

Display name: Super awesome text files

Logo:

Info tip:

Name: satxt

Edit flags

☑ Open is safe

☐ Always unsafe

Supported file types

At least one file type must be supported. Enter at least one file type; for example, ".jpg".

Supported file type

Content type: text/plain

File type: .satxt

Figure 15.2 The additional settings required for Documents Library and Removable Storage capabilities

ness account; individuals can't create apps that use these capabilities. *We will use these capabilities in this app, but for demonstration purposes, check the Documents Library, Music Library, Pictures Library, and Removable Storage capabilities.*

When you request access to the two restricted capabilities, it's typically because your app will have a specific type of file associated with it. Even apps with Documents Library access can't open just any old file; they need to specify the supported extensions in advance. This is done through the Declarations tab, as shown in figure 15.2.

You're filling out the capabilities information solely to enable access to the Documents Library and Removable Storage capabilities. Normally, you'd also make the app respond to file activation for the specified file types. You can find more information on file activation and working with associated files at http://bit.ly/Win8FileActivation.

15.1.3 Storage files and folders

When working with files from code, there are a few classes you'll need to understand. The classes used for Windows Store apps differ from those you may be familiar with on the desktop, because the definition of *file*, from an API standpoint, has been expanded to include resources as well as remote files.

The most important is the `StorageFile` class. This class, in the `Windows.Storage` namespace, represents a file, either remote or local. Many of the file operations in the Windows Runtime return or accept a `StorageFile` instance or a stream obtained from the instance.

Assuming the app has the correct permissions, the `StorageFile` class provides methods to copy files, copy and replace, delete, rename, move, open, and more. Above that, it also contains methods and properties to get metadata for the file, get the system-generated thumbnail, display name, path, full filename, and so on. You can even get a raw stream of data from a file or URI. This class will provide to you just about everything you need to work with a file and in a secure way that respects the Windows Store app permissions.

One thing the `StorageFile` doesn't represent is a folder. For that, you have the `StorageFolder` class. The `StorageFolder` class shares many of the same capabilities as the `StorageFile`, as well as functions to get lists of `StorageFile` instances from the folder.

Access to these folders, the actual locations of which may change from machine to machine and user to user, is provided through the `KnownFolders` class. The following listing shows how to use the `KnownFolders` class to gain access to three different file locations. Place the code in the `MainViewModel` class, replacing the empty placeholder methods that were already there.

Listing 15.4 Viewmodel code to list all files in certain folders

```
private async void UpdateChildren()
{
  var items = await CurrentFolder.GetItemsAsync();

  ChildItems.Clear();

  foreach (var item in items)
    ChildItems.Add(item);
}

public void LoadDocumentsLibrary()
{
  CurrentFolder = KnownFolders.DocumentsLibrary;
}

public void LoadMusicLibrary()
{
  CurrentFolder = KnownFolders.MusicLibrary;
}

public void LoadRemovableDevices()
{
  CurrentFolder = KnownFolders.RemovableDevices;
}
```

Get all items in folder

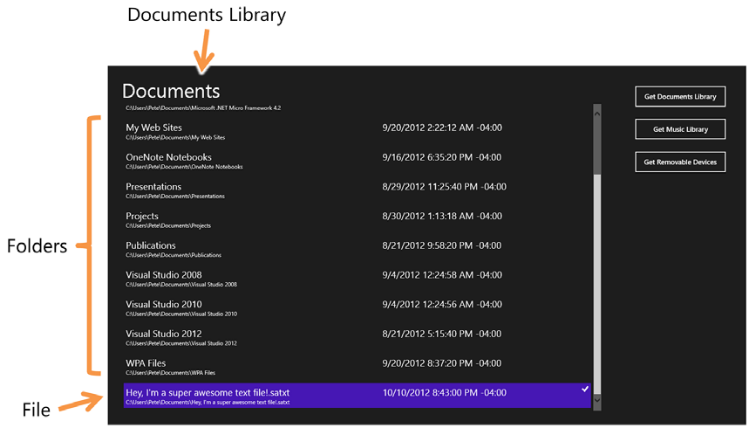
Clear old contents

Add items to collection

Use KnownFolders class

When you look at this code, it hits you just how easy it is (once you have permissions) to enumerate files and folders in a known location on the machine. Simply grab a reference to the folder using the `KnownFolders` class, then asynchronously call the `GetItemsAsync` method to get everything in that folder.

Figure 15.3 shows a shot of the app's screen with the contents of the documents library loaded up. My documents library on this test machine has only a single file but a bunch of folders.



Documents
C:\Users\Pete\Documents\Microsoft .NET Micro Framework 4.2

My Web Sites 9/20/2012 2:22:12 AM -04:00
C:\Users\Pete\Documents\My Web Sites

OneNote Notebooks 9/16/2012 6:35:20 PM -04:00
C:\Users\Pete\Documents\OneNote Notebooks

Presentations 8/29/2012 11:25:40 PM -04:00
C:\Users\Pete\Documents\Presentations

Projects 8/30/2012 1:13:18 AM -04:00
C:\Users\Pete\Documents\Projects

Publications 8/21/2012 9:58:20 PM -04:00
C:\Users\Pete\Documents\Publications

Visual Studio 2008 9/4/2012 12:24:58 AM -04:00
C:\Users\Pete\Documents\Visual Studio 2008

Visual Studio 2010 9/4/2012 12:24:56 AM -04:00
C:\Users\Pete\Documents\Visual Studio 2010

Visual Studio 2012 8/21/2012 5:15:40 PM -04:00
C:\Users\Pete\Documents\Visual Studio 2012

WPA Files 9/20/2012 8:37:20 PM -04:00
C:\Users\Pete\Documents\WPA Files

Hey, I'm a super awesome text file!.satxt 10/10/2012 8:43:00 PM -04:00
C:\Users\Pete\Documents\Hey, I'm a super awesome text file!.satxt

Figure 15.3 The app shown using the same item template for each storage item displayed in the list. The contents of the documents library are shown.

To create this UI, there's one piece left: the `ListView`. The list of items will include the name, the path to the item, and the date it was created. The following listing has the markup to put in MainPage.xaml in the spot you earlier reserved for the `ListView`.

Listing 15.5 MainPage.xaml `ListView` markup to create the UI

```
<ListView x:Name="ItemsList"
          Grid.Row="1"
          ItemsSource="{Binding ChildItems}">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Grid Margin="5">
        <Grid.Resources>
          <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="20" />
          </Style>
        </Grid.Resources>

        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="550" />
          <ColumnDefinition Width="300" />
        </Grid.ColumnDefinitions>

        <TextBlock Text="{Binding Name}"
                   Grid.Column="0"
                   Grid.Row="0"/>

        <TextBlock Text="{Binding DateCreated}"
                   Grid.Column="1"
                   Grid.Row="0"/>

        <TextBlock Text="{Binding Path}"
                  FontSize="12"
                   Grid.Column="0"
```

Bind to viewmodel property

Bind to properties of the item

```
                   Grid.ColumnSpan="2"
                   Grid.Row="1"/>

      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

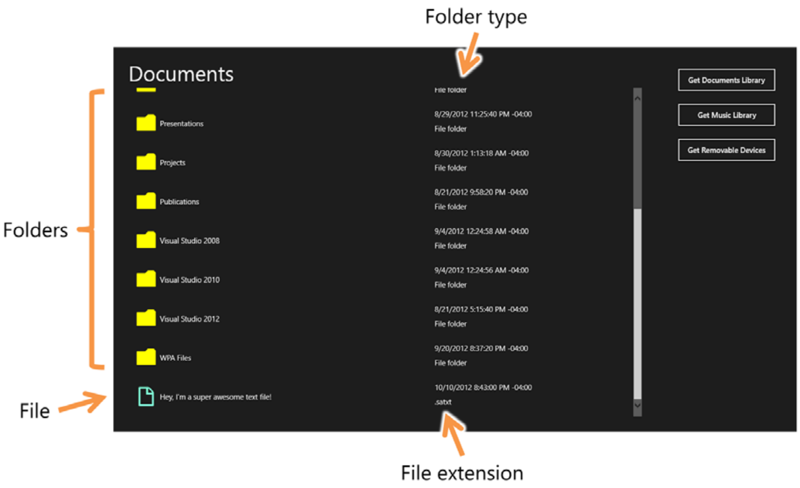
This code uses a plain item template that pulls the common fields provided by the `IStorageItem` interface. The `ListView` itself is bound to the collection on the viewmodel. If you run the app now and click one of the buttons, you'll get back a list of items. If you instead receive an exception, check to make sure you set up the capabilities and declarations in the manifest.

As you can see, it was really easy to list the files and folders and display their shared properties (name, date, and full path).

15.1.4 Using a data template selector

You have all the files and folders loaded and displayed in a list, but what if you wanted to display the storage items differently based on some criteria, for example, show folders differently from files? This seems like the perfect time to introduce to you the `DataTemplateSelector` class.

The `DataTemplateSelector` is a class you derive from to create logic that provides a reference to a `DataTemplate` based on the item passed in. The code could inspect a property of the item, or as you'll do here, make the decision based on the type of the item. The `ListView` and `GridView` (as well as all other `ItemsControl`-derived classes) have built-in support for working with a data template selector, so you'll use it to provide slightly different folder and file templates, as shown in figure 15.4.



Documents

	File folder
Presentations	8/29/2012 11:25:40 PM -04:00 File folder
Projects	8/30/2012 1:13:18 AM -04:00 File folder
Publications	8/21/2012 9:58:20 PM -04:00 File folder
Visual Studio 2008	9/4/2012 12:24:58 AM -04:00 File folder
Visual Studio 2010	9/4/2012 12:24:56 AM -04:00 File folder
Visual Studio 2012	8/21/2012 5:15:40 PM -04:00 File folder
WPA Files	9/20/2012 8:37:20 PM -04:00 File folder
Hey, I'm a super awesome text file!	10/10/2012 8:43:00 PM -04:00 .satxt

Figure 15.4 The `StorageFolder` and `StorageFile` items in the collection are rendered with different templates, selected through a custom `DataTemplateSelector` class.

Create a new folder named Selectors. Then, in that folder, create a class named `StorageItemDataTemplateSelector`. The following listing shows the source for this class.

Listing 15.6 The `StorageItemDataTemplateSelector` class

```
using System;
using Windows.Storage;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace DemoApp.Selectors
{
  public class StorageItemDataTemplateSelector : DataTemplateSelector
  {
    public DataTemplate FileTemplate { get; set; }
    public DataTemplate FolderTemplate { get; set; }

    protected override DataTemplate SelectTemplateCore(
          object item, DependencyObject container)
    {
      if (item is StorageFile)
        return FileTemplate;

      if (item is StorageFolder)
        return FolderTemplate;

      return base.SelectTemplateCore(item, container);
    }
  }
}
```

Inherit from DataTemplateSelector

Templates to pick from

Use FileTemplate for StorageFile instances

Use FolderTemplate for StorageFolder instances

The `StorageItemDataTemplateSelector` class overrides the important `SelectTemplateCore` method. This is the method that the runtime calls to determine which template to use. Inside that code, the method inspects the type. If it's a `StorageFile`, it returns whatever is in the `FileTemplate` property. Similarly, if it's a `StorageFolder`, it returns whatever is in the `FolderTemplate` property. If for some reason neither applies, it returns the default by deferring to the base functionality.

> **TIP** Data template selectors are generally useful throughout an app's UI. This is one use case. If you want to display a Windows Store-type UI in a `GridView`, with different-size blocks showing different types of content, a data template selector will help you get there.

The next listing has the new `ListView` markup, which includes the two different templates as well as the selector. Completely replace the previous `ListView` with this one.

Listing 15.7 Updated `ListView` markup to use the selector

```
<ListView x:Name="ItemsList"
          Grid.Row="1"
          ItemsSource="{Binding ChildItems}">
```

```
<ListView.Resources>
  <Style TargetType="TextBlock">
    <Setter Property="FontSize" Value="20" />
  </Style>

  <Style TargetType="TextBlock"
         x:Key="IconStyle">
    <Setter Property="FontSize" Value="40" />
    <Setter Property="FontFamily" Value="Segoe UI Symbol" />
    <Setter Property="Margin" Value="2" />
    <Setter Property="Foreground" Value="Aquamarine" />
    <Setter Property="VerticalAlignment" Value="Center" />
  </Style>
```

FileTemplate

```
  <DataTemplate x:Key="FileTemplate">
    <Grid Margin="5">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="550" />
        <ColumnDefinition Width="300" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>

      <TextBlock Text="&#xe160;"
                 Style="{StaticResource IconStyle}"
                 Grid.Row="0"
                 Grid.RowSpan="2"
                 Grid.Column="0"/>

      <TextBlock Text="{Binding DisplayName}"
                 VerticalAlignment="Center"
                 Grid.Row="0"
                 Grid.RowSpan="2"
                 Grid.Column="1"/>

      <TextBlock Text="{Binding DateCreated}"
                 Grid.Row="0"
                 Grid.Column="2"/>

      <TextBlock Text="{Binding FileType}"
                 Grid.Row="1"
                 Grid.Column="2"/>
    </Grid>
  </DataTemplate>
```

FolderTemplate

```
  <DataTemplate x:Key="FolderTemplate">
    <Grid Margin="5">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="550" />
        <ColumnDefinition Width="300" />
      </Grid.ColumnDefinitions>
```

```
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <TextBlock Text="&#xe188;"
                   Style="{StaticResource IconStyle}"
                   Foreground="Yellow"
                   Grid.Row="0"
                   Grid.RowSpan="2"
                   Grid.Column="0"/>

        <TextBlock Text="{Binding DisplayName}"
                   VerticalAlignment="Center"
                   Grid.Row="0"
                   Grid.RowSpan="2"
                   Grid.Column="1"/>

        <TextBlock Text="{Binding DateCreated}"
                   Grid.Row="0"
                   Grid.Column="2"/>

        <TextBlock Text="{Binding DisplayType}"
                   Grid.Row="1"
                   Grid.Column="2"/>
      </Grid>
    </DataTemplate>
  </ListView.Resources>

  <ListView.ItemTemplateSelector>
    <selectors:StorageItemDataTemplateSelector
            xmlns:selectors="using:DemoApp.Selectors"
            FileTemplate="{StaticResource FileTemplate}"
            FolderTemplate="{StaticResource FolderTemplate}" />
  </ListView.ItemTemplateSelector>
</ListView>
```

Template selector

Inline namespace declaration

Now when you run the app, the `ListView` will use the custom item template selector to choose the correct template for each item. Item template selectors can be as simple or as complex as you need for your apps, allowing you to create layouts that best represent the individual items being displayed.

15.1.5 Using file queries

So far, you've used the `StorageFolder` class to list all the items within that folder. Often, an app needs to filter that list of files. Sure, this could be done in code after the fact, but wouldn't it be great if you could make a single call, optimized by the OS, with the filter already in place?

You can, of course. The feature in WinRT that makes this possible is the query. The following listing shows how. Update the `MainViewModel` code with the code from this listing.

Listing 15.8 Listing files using a query

```
private StorageFolder _currentFolder;
public StorageFolder CurrentFolder
{
  get { return _currentFolder; }
  set
  {
    _currentFolder = value;
    NotifyPropertyChanged("CurrentFolder");
    //UpdateChildren();
    UpdateChildrenWithQuery();
  }
}

private async void UpdateChildrenWithQuery()
{
  var options = new QueryOptions(CommonFileQuery.OrderByDate,
        new string[] { ".satxt", ".docx", ".txt" });

  var result = KnownFolders.DocumentsLibrary.CreateItemQueryWithOptions(
        options);

  var items = await result.GetItemsAsync();

  ChildItems.Clear();

  foreach (var item in items)
    ChildItems.Add(item);
}
```

Old update call

New update call

Sort by date, and filter by extension

Create items query

Get items from query

Add items to collection

The code in this listing adds a new `UpdateChildren` method, this time called `UpdateChildrenWithQuery`. The query is built so that it sorts the data by date and filters the results to any files with the listed extensions. Note that because this is an item query, it will still return folders because the extension doesn't apply to them. If you want only files or only folders, you can use the appropriate methods to create a file query or a folder query instead of an item query.

Importantly, the query is also recursive; it will return all files and folders that meet the criteria, regardless of their depth in the tree. To avoid this, use the `FolderDepth` property of the query options object. For more information on this and other query options, see http://bit.ly/Win8QueryOptions.

15.1.6 Creating files and folders

Maybe the main purpose of your app is to create a data file to be used by other apps. Perhaps your app provides value by editing data created by other apps. Although listing files tends to be more common than writing files and creating folders, many apps do need to open or create individual files or even create subfolders.

For this next example, you'll create a folder and file using user-specified data. The `StorageFolder` class provides methods for creating both files and folders. As part of the creation process, you can specify naming collision options (error, overwrite, generate a name, and so on). The next listing has the required code for the `MainViewModel` class.

Listing 15.9 `MainViewModel` code to create a subfolder and file

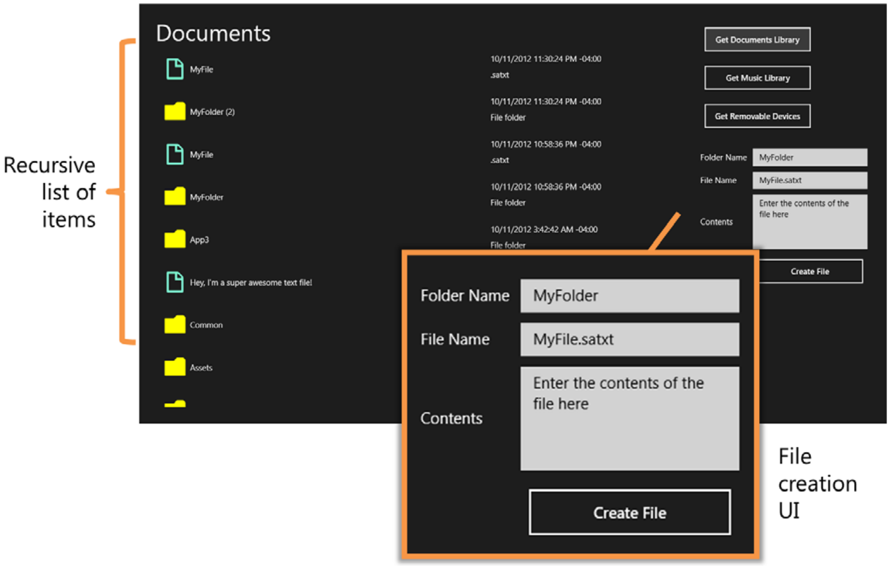
```
public async Task<StorageFile> CreateFile(
                    string subFolderName, string fileName)
{
  if (CurrentFolder != null)
  {
    var folder = await CurrentFolder.CreateFolderAsync(
            subFolderName,
            CreationCollisionOption.GenerateUniqueName);

    return await folder.CreateFileAsync(
            fileName,
            CreationCollisionOption.GenerateUniqueName)
            .AsTask();
  }
  else
  {
    return null;
  }
}
```

Create subfolder

Create file

Return null if no parent folder selected

This listing creates a folder and then a file inside the folder. The file has no contents. Instead, the file is returned as the result of this asynchronous method, to be filled by the calling code.



Recursive list of items

MyFile — 10/11/2012 11:30:24 PM -04:00 — .satxt
MyFolder (2) — 10/11/2012 11:30:24 PM -04:00 — File folder
MyFile — 10/11/2012 10:58:36 PM -04:00 — .satxt
MyFolder — 10/11/2012 10:58:36 PM -04:00 — File folder
App3 — 10/11/2012 3:42:42 AM -04:00 — File folder
Hey, I'm a super awesome text file!
Common
Assets

Folder Name: MyFolder
File Name: MyFile.satxt
Contents: Enter the contents of the file here

File creation UI

Figure 15.5 The new file-creation UI displayed alongside the list of files and folders returned from the query. Note that the file list is recursive; that's why you see multiple MyFile instances in there. Note also the MyFolder and MyFolder (2) entries. Those were folders created through this UI.

The collision options set for the two calls will result in numbered folders and files if run multiple times. You've likely seen this in Windows before: A copied file gets a (2) appended to the name. That's exactly what the `GenerateUniqueName` option does here. Of course, you could also have checked to see if the folder exists and, if so, simply used it rather than create it.

You'll need a simple UI to drive this. It will have entry fields for the folder and filename, a large `TextBox` for the contents, and then a button to create the folder and file. Figure 15.5 shows what it will look like when run.

The next listing shows the XAML for the UI. Place this markup right under the other buttons inside the `StackPanel` on the right of the page. Be sure the `Grid` is inside the `StackPanel` or the UI will get overlaid.

Listing 15.10 MainPage.xaml additional UI for creating files

```
<Grid Margin="0,20,0,0">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>

  <Grid.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="Margin" Value="5" />
      <Setter Property="VerticalAlignment" Value="Center" />
      <Setter Property="FontSize" Value="15" />
    </Style>
    <Style TargetType="TextBox">
      <Setter Property="Margin" Value="5" />
      <Setter Property="VerticalAlignment" Value="Center" />
      <Setter Property="FontSize" Value="15" />
    </Style>
  </Grid.Resources>

  <TextBlock Text="Folder Name"
             Grid.Column="0" Grid.Row="0"/>

  <TextBox x:Name="FolderName" Text="MyFolder"
           Grid.Column="1" Grid.Row="0"/>

  <TextBlock Text="File Name" Grid.Column="0" Grid.Row="1"/>

  <TextBox x:Name="FileName" Text="MyFile.satxt"
           Grid.Column="1" Grid.Row="1"/>

  <TextBlock Text="Contents"
             Grid.Column="0" Grid.Row="2"/>
```

Folder name

File name

```
  <TextBox x:Name="Contents"
           Text="Enter the contents of the file here"
           Height="100" TextWrapping="Wrap"
           IsSpellCheckEnabled="True"
           IsTextPredictionEnabled="True"
           Grid.Column="1" Grid.Row="2"/>

  <Button Content="Create File"
          HorizontalAlignment="Right"
          Grid.Row="3" Grid.Column="1"
          Click="CreateFileClick" />
</Grid>
```

File contents

Button to create file

There's nothing surprising about this UI, except for the fact that you're not using any data binding to hold the entered values. Why not? Mostly because it simplifies the code in this example and because you won't be using those entered values in any other place. The following listing shows that code in the code-behind.

Listing 15.11 MainPage.xaml.cs code to handle the new UI elements

```
private async void CreateFileClick(object sender, RoutedEventArgs e)
{
  string folderName = FolderName.Text.Trim();
  string fileName = FileName.Text.Trim();
  string contents = Contents.Text.Trim();

  var file = await _vm.CreateFile(folderName, fileName);

  if (file != null)
  {
    await FileIO.AppendTextAsync(file, contents);
  }
  else
  {
    var dlg = new MessageDialog("Unable to create file.");
    await dlg.ShowAsync();
  }
}
```

Call viewmodel to create file

Append the text to the file

Display message if creation failed

This code first grabs the values from the text entry fields on the screen. Then, it calls the viewmodel's `CreateFile` method, passing in only the folder and filename. If file creation succeeds, it then writes the supplied text to the file. If the creation fails (for example, if you forget to select a parent folder first), the code displays a message to the user.

The `FileIO` class is a nice helper class that makes writing to `StorageFiles` easier. If you don't have a `StorageFile` instance but do have a path or URI, you can use the `PathIO` class.

The example here also uses the `MessageDialog` class to display a message. The `MessageDialog` is the WinRT equivalent of the old `MessageBox` class. Use it only when you need to display a message that requires attention. For other transient messaging (like success), use toast or an onscreen element to display the update.

Windows Store apps have a great amount of support for programmatically listing and accessing files and folders in a secure way. If you want to gain easy access to the commonly used folders on the PC, you can use the `KnownFolders` class. But to access the individual folders, such as the music library or documents library, you'll need to first set up the appropriate declarations in the app manifest.

The folders surfaced by the `KnownFolders` class are `StorageFolder` instances. The `StorageFolder` class provides access to create, update, and delete files and subfolders. The `FileIO` class will also help you with those tasks. The `StorageFolder` class also provides functions to list all the files, all the folders, or a combination of the two. Beyond that, it provides the ability to create file queries that recursively, if desired, return items from the folder based on criteria you specify.

So far, you've used only the `KnownFolders` class to access files from code. In the next section, we'll look at URI formats that may be used from code or from markup.

Other file access options

The `FileIO` and `PathIO` classes aren't the only ways to read and write bytes. The `StorageFile` class provides access to the raw stream for both reading and writing, which, although slightly more complex, is generally more flexible.

Additionally, `StorageFile` has methods for loading a file from a path or URI and also for generating thumbnails for the file. This last capability, exposed through the `GetThumbnailAsync` method, is great for generating thumbnails for all file types, not just images. Because it uses the OS, the thumbnails are cached and are generated very quickly.

Thumbnail generation is a method and not a property; you'll need to provide your own model objects with the thumbnail rather than bind to instances of `StorageFile` directly. The MVVM pattern can certainly help you there.

You can find out more about the `StorageFile` class on MSDN at http://bit.ly/Win8StorageFile. While there, look at the other classes in the `Windows.Storage` namespace, because many are optimized for specific scenarios like bulk access of file properties, creating files from streams, virtualization, and the like.

In particular, the `FileInformationFactory` is what you want to look at if you need to pull a large list of thumbnails in a virtualized and high-performing way.

15.2 URI formats

Windows 8 introduces a number of important URI formats that may be used to access data throughout your app. The use of named URIs enables file access directly in XAML (or HTML) without requiring any running code.

So far, you've seen the `ms-appx` scheme used primarily to reference images in other examples, but there are several other schemes, as shown in table 15.1.

Table 15.1 Known URI formats for Windows 8 apps

Example URI	Description
`http://<domain>/<file>`	Accesses a file from the web, just as you would from the browser. There are no cross-domain or cross-scheme restrictions as in some other technologies like Silverlight or Flash.

`ms-appx://<folder>/<file>`
or
`<folder/file>`
	Loads a file in a path relative from the current page or code. If used in a XAML page in a subfolder in the project, for example, the path is expressed relative to that page.

`ms-appx:///<folder>/<file>`
or
`/<folder>/<file>`
	References a file root-relative from the installation folder or project root.

`ms-appx:///<classlib>/<folder>/<file>`	Opens a resource embedded in a referenced class library.
`ms-appdata:///local/<folder>/<file>`	References a file stored in the app's local data store. This is typically used for state that's specific to the machine. This is also a good place to store app-specific data that's not useful cross-machine.
`ms-appdata:///roaming/<folder>/<file>`	References a file stored in the app's roaming data store. This is synchronized between PCs and is associated with the account of the user logged in when accessed. This is the best place to store app-specific data that the user may want to share between machines.
`ms-appdata:///temp/<folder>/<file>`	References the local machine's temp folder for this app. Use this when you need to write disposable intermediate files.

When passing URIs to methods in the Windows Runtime, you need to use the full scheme, not a relative URI. To keep things simple, I always use the scheme, such as `ms-appx`.

These URI formats will be especially useful in markup. When working from code, you can use these URIs in functions designed to work with them, but by far the safest, and most flexible, approach for opening files is to use the file picker.

Storing app settings

You may be tempted to store app settings in a file. But if the data is designed in such a way that it can be represented by one or more name and value pairs, you can instead use the `ApplicationData` class.

(continued)

Through this class, you can designate groups of settings as local, roaming, or temporary, as well as version the settings data to make upgrades easier. We'll cover more on the `ApplicationData` class, and the Settings charm, in chapter 22.

15.3 Working with file pickers

All of the file access in this chapter has been completely controlled by code or markup. What happens when you want to put the user in control of picking the files? Does WinRT have an equivalent to the venerable `FileOpenDialog` we've grown to love?

WinRT not only has an equivalent, but it has also taken the concept of opening a file and made it extensible so that any app can plug into it to serve as the source of a file. That means you're not limited to files on the local machine or in specific locations. You're not even limited to files that actually exist! The logical file could have come from the web, from a compilation of fields in a local database, from an app's private data store, or as the result of a computation. If another app can serve up the request with a `StorageFile`, you can use it.

There are three pickers, all of which work similarly: the `FileOpenPicker`, the `FileSavePicker`, and the `FolderPicker`. In this section, we'll first look at how to use the `FileOpenPicker` to open files from other apps. Then, you'll implement your own `FileOpenPicker` source functionality so that you can serve files to other apps.

15.3.1 Using the file open picker

The `FileOpenPicker` builds on your knowledge of the `StorageFile` class. Once you use the picker to obtain a `StorageFile` instance, you can work with it just as you have in the previous sections in this chapter.

For our example, you'll put a button on the screen that will let you open the picker. The following listing has the button markup to place under the other buttons in the `StackPanel` on the right of the main page.

Listing 15.12 MainPage.xaml UI for file open functionality

```
<StackPanel>
  <Button Content="Get Documents Library"
          Click="GetDocumentLibraryClick"/>
  <Button Content="Get Music Library"
          Click="GetMusicLibraryClick"/>
  <Button Content="Get Removable Devices"
          Click="GetRemovableDevicesClick"/>

  <Button Content="Open File Picker"
          Click="OpenFilePickerClick"/>
  ...
</StackPanel>
```

Existing buttons

New button

The markup adds a single button and wires it up to the event handler. The event handler itself is where all the magic happens. To implement it, put the code in the next listing into the MainPage.xaml.cs code-behind.

Listing 15.13 Code-behind to open the file

```
private async void OpenFilePickerClick(object sender, RoutedEventArgs e)
{
  var picker = new FileOpenPicker();

  picker.ViewMode = PickerViewMode.Thumbnail;
  picker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
  picker.FileTypeFilter.Add(".jpg");
  picker.FileTypeFilter.Add(".png");
  picker.FileTypeFilter.Add(".satxt");
  picker.FileTypeFilter.Add(".txt");
  picker.FileTypeFilter.Add(".docx");

  var file = await picker.PickSingleFileAsync();

  if (file != null)
  {
    var dlg = new MessageDialog(file.DisplayName + " was picked.");
    await dlg.ShowAsync();
  }
  else
  {
    var dlg = new MessageDialog("No file picked.");
    await dlg.ShowAsync();
  }
}
```

Create picker

Show thumbnails, not a list

Set file types you're interested in

Show picker

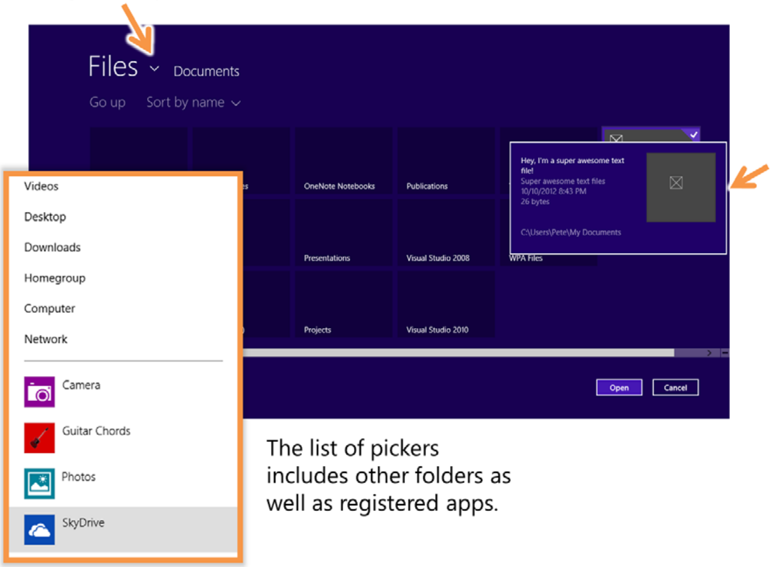
A file was picked

This code creates an instance of the file open picker, sets it to display thumbnail images rather than a list, and sets a suggested start location. The start location is a suggestion only, because the picker will remember the user's last location and almost always start from there. The suggestion only comes into play the first time the picker is used.

Then, once everything is set up, the `PickSingleFileAsync` method is called. This method, as the name suggests, enables picking a single file. There's also a `PickMultipleFilesAsync` method, which allows more than one file to be picked.

If a file was picked, you display a message with the display name of the file. If not, you display a simple "No file picked" message. At the point those messages are displayed, you have access to the `StorageFile` returned from `PickSingleFileAsync`, so you could write to it, read from it, and so on.

You'll need to add a `using` statement for the `Windows.Storage.Pickers` namespace for the code in listing 15.13 to compile. That namespace is the one that contains the `FileOpenPicker` and related classes. Once you do, run the app and click the button to open the picker. You'll see something similar to figure 15.6. In this case, I've hovered (with the mouse) over a file with the .satxt extension, the extension registered to our app. You can see that I didn't provide an icon or any preview information for it.


Files
Documents
Go up
Sort by name
OneNote Notebooks
Publications
Presentations
Visual Studio 2008
WPA Files
Projects
Visual Studio 2010
Hey, I'm a super awesome text file!
Super awesome text files
10/10/2012 8:43 PM
26 bytes
C:\Users\Pete\My Documents
Open
Cancel
Videos
Desktop
Downloads
Homegroup
Computer
Network
Camera
Guitar Chords
Photos
SkyDrive

Click arrow to see list of all registered pickers.

File type registered to our app

The list of pickers includes other folders as well as registered apps.

Figure 15.6 The file open picker in use by our app. The called-out list shows available pickers.

Additionally, if you click the little downward arrow or chevron to the right of Files in the displayed picker, you'll be presented with a list of file locations as well as apps that are registered as file open pickers.

You can also save files using the `FileSavePicker`. The approach is similar, so I won't duplicate it here. But let's take a look at how to create your own file open picker.

15.3.2 *Implementing the file picker source contract*

When your app registers itself as a file open picker, it must provide a UI to allow the user to pick a file. The app doesn't own the entire UI, however, just the horizontal band under the heading and above the Open and Cancel buttons. Everything else is owned by Windows to provide a consistent experience.

In the root of the project, add a new blank page named `FileOpenPickerPage`. The markup for this page is shown here.

Listing 15.14 The picker page

```
<Page
  x:Class="DemoApp.FileOpenPickerPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:DemoApp"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <GridView x:Name="FileList">
      <GridView.ItemTemplate>
        <DataTemplate>
          <Grid HorizontalAlignment="Left" Width="250" Height="100">
            <StackPanel VerticalAlignment="Bottom"
                        Background="{StaticResource
                        ➥ ListViewItemOverlayBackgroundThemeBrush}">

              <TextBlock Text="{Binding DisplayName}"
                         Foreground="{StaticResource
                         ➥ ListViewItemOverlayForegroundThemeBrush}"
                         Style="{StaticResource TitleTextStyle}"
                         Height="60"
                         Margin="15,0,15,0"/>

              <TextBlock Text="{Binding FileType}"
                         Foreground="{StaticResource
                         ➥ ListViewItemOverlaySecondaryForegroundThemeBrush}"
                         Style="{StaticResource CaptionTextStyle}"
                         TextWrapping="NoWrap"
                         Margin="15,0,15,10"/>
            </StackPanel>
          </Grid>
        </DataTemplate>
      </GridView.ItemTemplate>
    </GridView>
  </Grid>
</Page>
```

Display name

File type (extension)

This listing has a simplified `GridView` without any real extras. The item template is based on the 250 px item template included in the StandardStyles.xaml file. I simplified it a bit to remove the image thumbnails (because the thumbnails for files must be generated in code, and we're not doing that) and to make it so more of the items would fit on the screen at once.

How is this screen invoked? It's through a special override in app.xaml.cs, as shown in the following listing.

Listing 15.15 app.xaml activation code

```
protected override void OnFileOpenPickerActivated(
                             FileOpenPickerActivatedEventArgs args)
{
  var page = new FileOpenPickerPage(args);

  Window.Current.Content = page;
  Window.Current.Activate();
}
```

Create page, passing args

Activate page

This method is called when the app is loaded as a file open picker. All the code does is instantiate the `FileOpenPickerPage` and activate it. When creating the page, it passes in the `args` property.

When creating an instance of this page, you must pass in the arguments provided at app activation. It's these arguments that you use to add individual files. The next listing shows the constructor with the arguments, as well as the code that synchronizes the collections. More on that after the code.

Listing 15.16 File picker page code-behind

```
using System;
using Windows.ApplicationModel.Activation;
using Windows.Storage;
using Windows.Storage.Pickers.Provider;
using Windows.Storage.Search;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace DemoApp
{
  public sealed partial class FileOpenPickerPage : Page
  {
    FileOpenPickerActivatedEventArgs _args;

    public FileOpenPickerPage(FileOpenPickerActivatedEventArgs args)
    {
      this.InitializeComponent();

      _args = args;

      LoadFiles();

      if (_args.FileOpenPickerUI.SelectionMode ==
                       FileSelectionMode.Multiple)
        FileList.SelectionMode = ListViewSelectionMode.Multiple;
      Else
        FileList.SelectionMode = ListViewSelectionMode.Single;

      _args.FileOpenPickerUI.Title = "Pete's cool files";


      FileList.SelectionChanged +=
                    FileList_SelectionChanged;
      _args.FileOpenPickerUI.FileRemoved +=
                    FileOpenPickerUI_FileRemoved;
    }

    void FileOpenPickerUI_FileRemoved(FileOpenPickerUI sender,
                                      FileRemovedEventArgs args)
    {
      if (FileList.SelectionMode == ListViewSelectionMode.Multiple)
      {
        foreach (StorageFile file in FileList.SelectedItems)
        {
          if (file.Path == args.Id)
          {
            FileList.SelectedItems.Remove(file);
            break;
          }
        }
```

Reference to args

Load files

Set selection mode

Set page subtitle

Wire synchronization events

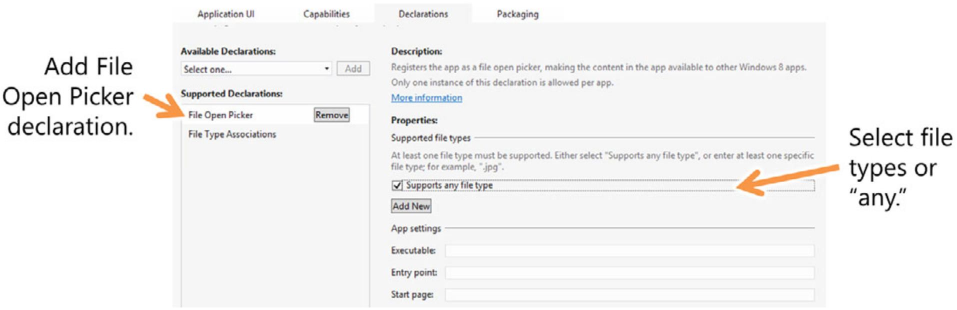
Synchronize external changes

```
      }
      else
      {
        if (FileList.SelectedItem != null)
        {
          if (((StorageFile)(FileList.SelectedItem)).Path ==
                       args.Id)
            FileList.SelectedItem = null;
        }
      }
    }
                                                                Synchronize
    void FileList_SelectionChanged(object sender,            <─ internal changes
                                   SelectionChangedEventArgs e)
    {
      foreach (StorageFile file in e.AddedItems)
        _args.FileOpenPickerUI.AddFile(file.Path, file);

      foreach (StorageFile file in e.RemovedItems)
        _args.FileOpenPickerUI.RemoveFile(file.Path);
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {   }

    private async void LoadFiles()
    {
      var options = new QueryOptions(
                CommonFileQuery.OrderByDate,
                new string[] { ".jpg", ".png" });
                                                                Load files,
      var result = KnownFolders.PicturesLibrary                 only
              .CreateFileQueryWithOptions(options);          <─

      var files = await result.GetFilesAsync();

      FileList.ItemsSource = files;                 <─ Bind list
    }
  }
}
```

This is one case where handling the interaction of code and UI without using binding is much easier than with binding. In order to be a file picker, the app needs to respond to changes in the dictionary of selected files, as well as add and remove files from that dictionary based on user selection. You could find a way to do this using MVVM, but sometimes the effort gets in the way of the end goal, as it would here.

The final step in implementing `FileOpenPicker` functionality is to declare the capability in the Declarations tab of the appx manifest. First, add the declaration, and then decide which file types you will support. If you support any file type, select that option in the properties. Otherwise, be sure to specify the exact types your app provides. Figure 15.7 shows the setting in place.

To test the functionality, build and deploy the app. But instead of running it, switch over to an app that uses the file picker. SkyDrive is a simple one to use. Then,


Application UI
Capabilities
Declarations
Packaging
Select one...
Add
File Open Picker
Remove
File Type Associations
More information
Add New

Add File Open Picker declaration.

Available Declarations:

Supported Declarations:

Description:
Registers the app as a file open picker, making the content in the app available to other Windows 8 apps.
Only one instance of this declaration is allowed per app.

Properties:
Supported file types
At least one file type must be supported. Either select "Supports any file type", or enter at least one specific file type; for example, ".jpg".
☑ Supports any file type

App settings
Executable:
Entry point:
Start page:

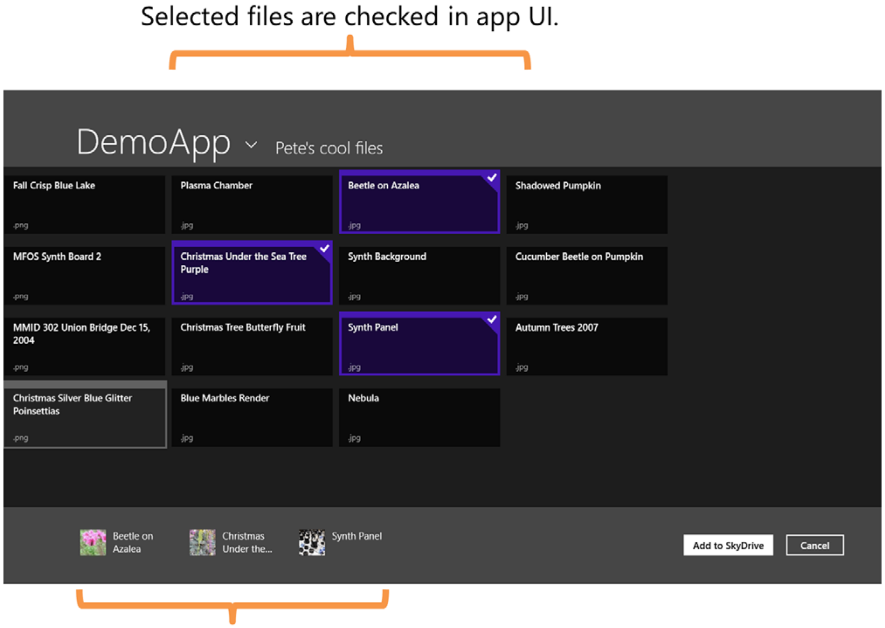
Select file types or "any."

Figure 15.7 The File Open Picker declaration showing that our app provides picker functionality for any file type

open the app bar and choose the option to upload a file. You'll then be presented with the file picker. Click the little down arrow to the right of the title and pick your app as the source. You should see something that looks like figure 15.8 but with whatever files you have in your pictures library.

The file picker system in Windows 8 is also a great way for apps to provide functionality to other apps. Now apps don't need to know the source of the file; they simply get the file object and work on it. This makes it possible for you to pick documents from SkyDrive, photos from Flickr or Facebook, and much more.

Selected files are checked in app UI.


DemoApp
Pete's cool files
Fall Crisp Blue Lake
.png
Plasma Chamber
.jpg
Beetle on Azalea
.jpg
Shadowed Pumpkin
.jpg
MFOS Synth Board 2
.png
Christmas Under the Sea Tree Purple
.jpg
Synth Background
.jpg
Cucumber Beetle on Pumpkin
.jpg
MMID 302 Union Bridge Dec 15, 2004
.png
Christmas Tree Butterfly Fruit
.jpg
Synth Panel
.jpg
Autumn Trees 2007
.jpg
Christmas Silver Blue Glitter Poinsettias
.png
Blue Marbles Render
.jpg
Nebula
.jpg
Beetle on Azalea
Christmas Under the...
Synth Panel
Add to SkyDrive
Cancel

File picker UI mirrors app file selection.

Figure 15.8 Our app as a file open picker source. You can see that it's working because the list of files at the bottom of the screen properly reflects the selections made in our app UI.

For more information on consuming and creating file pickers, please see this MSDN page at http://bit.ly/Win8Pickers.

Using the SQLite database engine

Although I encourage you to use files whenever possible, because they make sharing and other contracts easier to work with, many apps could make better use of a database.

Windows 8 apps can use a local database engine, as long as that engine respects Windows Store rules. One such database is SQLite, which can be downloaded from http://sqlite.org/.

Because SQLite is implemented in C++, you'll need to provide separate app packages for each supported architecture. (The version included with ARM will be different for 64-bit Intel architecture, for example.)

At the time of this writing, the packaging and deployment for SQLite were still being finalized. Check on their site for more information about using SQLite with Windows Store apps.

15.4 Summary

File operations are some of the most fundamental services that any OS or platform can provide for app developers. Windows 8 supports all the basic file IO you'd expect but adds to that the ability to work natively with files from nontraditional sources. The `StorageFile` class and its related `IStorageFile` interface are flexible enough that the file could come from a website, a local folder, or another app on the system.

When you do want to work with local files, the `StorageFolder` class provides easy access to folders on the local machine. For a quick way to get a reference to the common folders such as the documents library, you can use the `KnownFolders` class, which exposes each of those common locations as `StorageFolder` instances. Once you have a storage folder, you can enumerate its items using either straight collections or the more flexible approach of file queries. By using queries, you let the OS handle filtering and sorting in the most efficient way possible.

When you want to access these locations from markup, you can use the set of URIs for known locations, websites, or even protocols provided by other apps.

The extensibility story in Windows 8 includes even the loading of files. Apps can register themselves as `FileOpenPicker` or `FileSavePicker`, thereby serving as the source of `StorageItem` instances for other apps. The files could be located anywhere or even be simply records in a database or something generated at runtime. This type of flexibility makes it possible for third-party app developers to provide value not only to their app users but to other apps in the system, making the whole platform better in the process.

As was also the case when I covered networking, there were a number of asynchronous operations in the file examples in this chapter. The next chapter covers async in detail so you can better understand how to use asynchronous methods as well as create your own.

16

Asynchronous everywhere

This chapter covers

- Working with asynchronous operations
- Using `async` and `await`
- Using .NET `Task` and the WinRT `IAsync*` approaches
- Converting between WinRT `IAsync*` and `Task`

How many times have you used an application that became unresponsive, if only for a few seconds? Starting in Windows Vista, those applications would frost over and respond to clicks by shifting themselves a couple pixels on the screen. More often than not, those applications were executing some long-running function or were waiting for a resource to be freed. For example, my data drive is a low-power spinning rust drive, and it tends to be aggressive about spinning down. Sometimes, when I go to save a document in Microsoft Word, Word will frost over for a couple of seconds as the drive spins up and the data can be saved.[1]

---

[1] My apps and Windows drive is on a big SSD that's always in use and always responsive. If it starts requiring spin-up time, I'm in serious trouble.

In most UI frameworks, Windows applications have the concept of a UI thread which is where all interaction with the user happens. Windows Forms, WPF, and Silverlight drove that concept home to many by throwing an exception if code on a background thread attempted to access objects on the UI thread. When that thread is locked or busy for too long, you get an unresponsive application and the manifestations mentioned earlier.

Over the past years, asynchronous operations have become a fact of life for developers. I blame Ajax[2] and JavaScript. What's really to blame here is the expectations of the user. There was a time when it was perfectly acceptable to have your application lock up and be unresponsive during a long calculation or other operation. From day one, every GUI has had an hourglass, clock, or watch pointer to make this seem normal. Many apps would hang when doing something as simple as printing.

But then we started adding in animation, and not just the sparkly animated GIFs made popular on GeoCities and Angelfire[3] but animation that is useful and conveys information such as context—for example, the expansion of an element when a mouse hovers over it, or the sliding in of an element so that it doesn't simply jump out at you like a monster in a cheap haunted house carnival ride.

Operations that caused the animation to stutter or skip suddenly became a real problem. Around the same time, we started seeing the ubiquity of discrete graphics cards (or graphics sections of modern CPUs) combined with main processors with much better baked-in support for additional threads and processes. Together, these were able to better carry out background operations without tying up the UI. The animation could be offloaded but so could the longer-running tasks that were tying up the UI in the first place.

One of the longest latency activities is networking, so browsers and Silverlight tackled those and ensured they were required to be asynchronous calls. For browser desktop technologies like WPF, the recommended approach was to use asynchronous code. But, because WPF and Windows Forms didn't require it, most desktop apps weren't written using asynchronous code. The result was clear: If it wasn't required, it didn't happen, because it was hard.

The straw that broke the camel's back was touch interaction. Unlike mouse interaction, touching a UI is such an immediate and deep interaction that any sort of lag in responsiveness becomes immediately noticeable. When you put apps that rely on touch on a tablet designed for touch, using a relatively low-power ARM processor, any and all lag becomes immediately noticeable. Importantly, users who see the lag don't just think "Wow, that app stinks" but also think "Wow, this tablet stinks" or "This operating system stinks." So, Microsoft made the decision to make anything even remotely slow into an asynchronous API.

---

[2] And you too, jQuery! I know what game you're playing.

[3] If you're too young to remember that colorful period in internet history, or you've simply blocked it out, check out the Geocities-izer at http://wonder-tonic.com/geocitiesizer/ or Cornify at http://www.cornify.com/.

Throughout the book I've been using asynchronous operations but without any detailed explanation. In this chapter, we'll take a look at how WinRT, .NET, and C# work together to make asynchronous calls work with a minimal amount of code. We'll first look at the reasons why asynchronous code is important. From there, we'll start with the Windows Runtime approach to asynchronous code and introduce the `async` and `await` keywords, together a key feature for the simplification of asynchronous calls. We'll also look at how to handle progress reporting and cancellation of long-running operations. Once we finish up with the WinRT approach, we'll look at how .NET has expanded the use of `Task` in .NET 4.5. Taking a similar approach to what we did with WinRT, you'll see how to use `Task` and how to cancel operations. We'll wrap up with information on how to convert between the WinRT `IAsync*` approach and the .NET `Task` approach.

But first, you need to understand why we're even worrying about asynchronous operations and why it merits a chapter all its own.

16.1 Why asynchronous is important

Synchronous functions have been, for a very long time, the norm in programming. In this type of function, the calling code must wait until every instruction is complete before the function will return. It's just naturally the way code is executed on a processor. In contrast, asynchronous functions are those that return immediately after kicking off some internal process on another thread. Figure 16.1 shows the difference.

The benefit of asynchronous code is that the calling code regains control much more quickly. This means, in the case of a UI thread, that it can process UI interaction and other high-priority tasks without being blocked by a long-running process.

Writing good asynchronous code can be a real hair-puller. Like dealing with thread synchronization in a critical application or learning recursion for the first time, it's

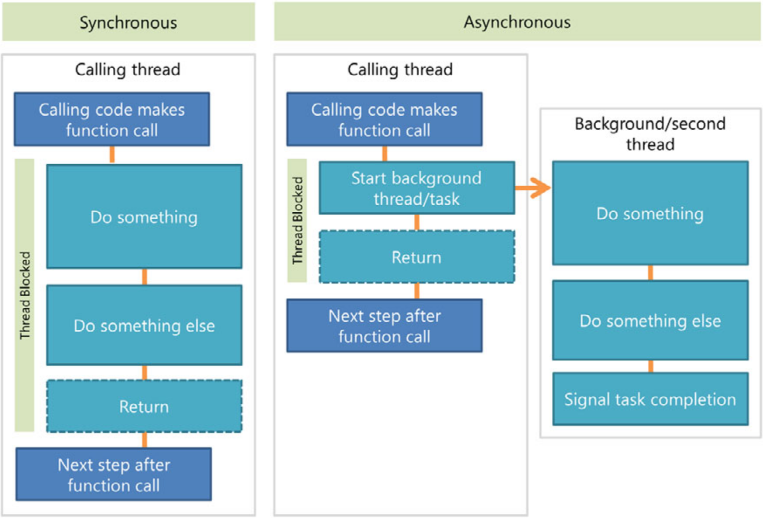

Synchronous
Calling thread
Calling code makes function call
Thread Blocked
Do something
Do something else
Return
Next step after function call
Asynchronous
Calling thread
Calling code makes function call
Thread Blocked
Start background thread/task
Return
Next step after function call
Background/second thread
Do something
Do something else
Signal task completion

Figure 16.1 Synchronous and asynchronous code compared. Notice how the calling thread regains control much earlier in the asynchronous version.

not something many look back on fondly. Writing asynchronous code is also important enough that it isn't something you can foist onto the new kid on your team like you do all those field validations and other boring code.

All .NET languages and platforms have supported asynchronous calls for quite some time, but for many of us, Silverlight was our first real exposure to asynchronous function calls in .NET. Until Silverlight, asynchronous code was never a platform-imposed requirement. Luckily, Silverlight kept this requirement to a single area: networking. But that single area was enough to make our code complex and to make many turn to a number of helper approaches such as Reactive Extensions (RX) and `Task`s from the Task Parallel Library (TPL). Most developers fought it and very few embraced it because, quite simply, it was a pain in the butt to deal with in all but the simplest of scenarios.

For these reasons, the Windows Runtime greatly expands on the use of asynchronous APIs. Instead of limiting asynchronous calls to only the most egregious offenders (networking), they are used any time there may be even slight latency in the response. You'll find them in networking, of course, but also in file access, pickers, stream manipulation, device enumeration, image encoding, and more. We're "all in" on async, baby!

> **NOTE** The bar for deciding when to make a method asynchronous was decided by the team implementing the Windows Runtime. A very quick 50 milliseconds (1/20th of a second) is the quoted number on blogs such as http://bit.ly/WinRTAsyncBackground, but that specific number is a design detail that's only a guideline, because performance will vary greatly between different classes of machines. In addition, it's erroneous to say that simply making functions asynchronous will guarantee good performance, because developers are typically very adept at finding ways to tie up the CPU.

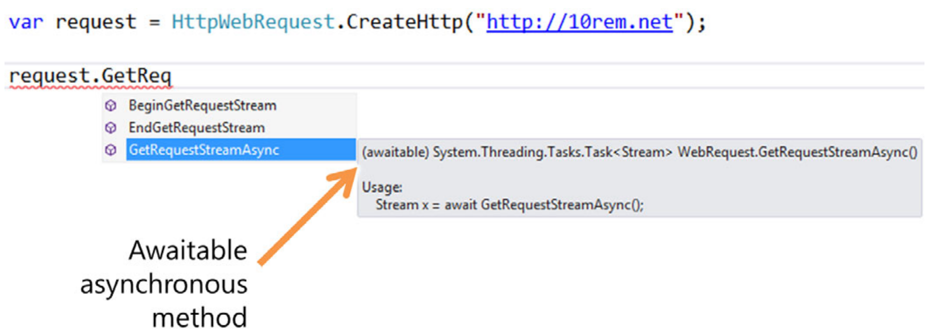
You'll be forgiven for thinking this expansion of the use of asynchronous APIs will make your code look like some horrible Rube Goldberg contraption of calls and callbacks (figure 16.2), synchronization blocks, and state management. It won't look like that because of help baked into C#.

Argonne
The Argonne Rube Goldberg
Machine Contest
GALORE

Figure 16.2 A group of developers at a convention dedicated to people writing interdependent multistep asynchronous code without the `await` keyword (Source: Argonne National Laboratory, CC license)[4]

---

[4] Aerial view of the "Argonne Rube Goldberg Machine Contest 2010," www.flickr.com/photos/argonne/4435710973/.



```
var request = HttpWebRequest.CreateHttp("http://10rem.net");

request.GetReq
```

(awaitable) System.Threading.Tasks.Task<Stream> WebRequest.GetRequestStreamAsync()

Usage:
  Stream x = await GetRequestStreamAsync();

Figure 16.3 The `GetRequestStreamAsync` method is an awaitable method as shown in C# IntelliSense. In this case, I'm using a networking class from the `System.Net.Http` namespace.

The latest version of C# introduced the `await` and `async` keywords (more on both shortly) to make it as easy to work with asynchronous code (`Task` or the `IAsync*` interfaces in WinRT) as it is to work with normal functions. Any asynchronous function that follows the patterns that enable the use of the `await` keyword is called an *awaitable function*.

Asynchronous functions in WinRT and .NET 4.5 have the suffix `Async` appended to their name. (This is an easily broken naming convention that isn't enforced by the compiler unless you're running code analysis/FXCop rules, so take care when writing your own code.) In C# IntelliSense you'll also see them marked as `awaitable`, as shown in figure 16.3.

It's worth noting that you'll run across two main types of `awaitable` functions. First, the Windows Runtime defines functions that return the interfaces `IAsyncAction`, `IAscyncActionWithProgress`, `IAsyncInfo`, `IAsyncOperation`, and `IAsyncOperationWithProgress` instances, collectively referred to by the shortcut `IAsync`*. Second, the .NET code you'll run in addition to WinRT tends to define functions that return instances of the `Task` class from the Task Parallel Library. Both are easy to work with and support the `await` keyword. But the `Task` class was defined in .NET before the creation of the Windows Runtime and is specific to .NET code. The interface-based approach in WinRT was created to support multiple languages.

16.2 Working with IAsync* WinRT methods

The Windows Runtime was built with the concept of asynchronous operations. It was also built with the idea that these asynchronous operations must work not only for C# but also for JavaScript, C++, and any number of other, very different languages. For those reasons, the team didn't take a dependency on the TPL in .NET but instead created an interface-based approach consistent with the rest of the Windows Runtime.

The `Windows.Foundation` namespace contains the types and interfaces required to support asynchronous functions in WinRT. Every asynchronous operation implements, at a minimum, the `IAsyncInfo` interface. Like most programming problems, there's an easy but limited way to do `async` and a more complex but capable way.

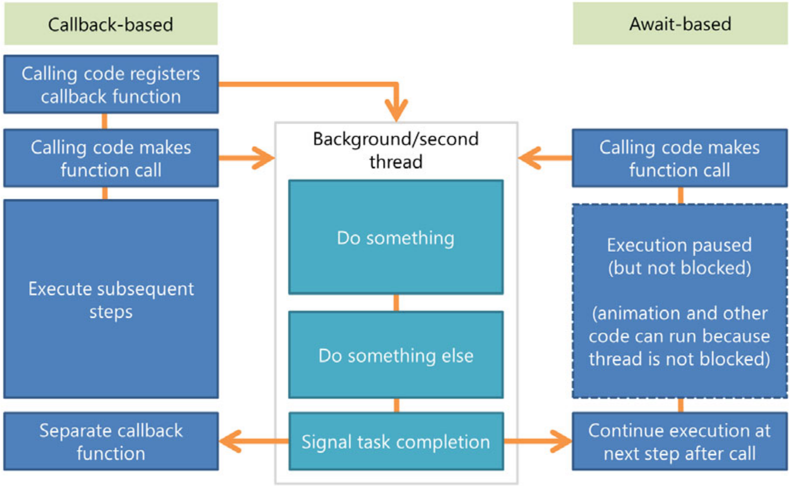

Callback-based
Await-based
Calling code registers callback function
Calling code makes function call
Execute subsequent steps
Separate callback function
Background/second thread
Do something
Do something else
Signal task completion
Calling code makes function call
Execution paused (but not blocked)
(animation and other code can run because thread is not blocked)
Continue execution at next step after call

Figure 16.4 The callback approach compared to the await-based approach. The primary difference is await-based code doesn't use an explicit callback and therefore looks almost identical to synchronous code. In the callback approach, it's up to the developer to synchronize the execution steps. In the await-based approach, the compiler handles keeping execution steps in order.

In this section, we'll look at the different approaches for using the WinRT asynchronous APIs—that is, the ones that return `IAsync*` interfaces. I'll cover the easiest approach and then look into what's needed to check progress or cancel an asynchronous operation.

16.2.1 async and await: the simplest approach

The biggest problem with working with asynchronous functions is how to deal with getting the result from the call. In traditional callback-based code, you had to have a callback function or event handler that would catch the return. This got pretty nasty when you had nested calls. What you really want, in most cases, is a simple way to wait for the result but without tying up the calling thread. Figure 16.4 shows the callback approach versus the ideal approach.

The simplest, and most commonly used, approach is to use the `await` keyword. A function that works with this approach is called an awaitable function. To make a call to an awaitable function, it must be used in a function marked as asynchronous`,` using the `async` keyword. Then all you need to do is use this keyword and call it like any normal synchronous function, as shown here.

Listing 16.1 Using the `await` keyword to perform a network operation

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  LoadFeedUsingAwait();
}
```

Function calls will go here

```
private async void LoadFeedUsingAwait()
{
  var client = new SyndicationClient();

  var uri = new Uri("http://feeds.feedburner.com/PeteBrown");
  var feed = await client.RetrieveFeedAsync(uri);

  Debug.WriteLine(feed.Title.Text);
}
```

Method marked as async

Async call using await

The code in the listing goes in the code-behind for the main page on a new application. Notice that you don't name the function `LoadFeedAsync`. That's because it's a void function, not a function that returns a `Task` or `IAsync*` interface. To be consistent with the platform-naming conventions, only name the function with the `Async` suffix if it is an awaitable function, returning an `async` type.

Keep the application open, because you'll try out a few other approaches before this chapter is through. Run the application and (on a single-display system) switch back to your IDE and look at the output window. You should see the title of my blog displayed there. On a dual-display system, the Modern Style app will show up on one display and the IDE on the other, so no switching is required.

TIP

Everyone knows that if it's printed in a book, it immediately has capital-A Authority, so here's my little gift to you. Copy this paragraph and send it to your manager:

Dear developer bosses of the world: Modern Style development on x86/x64 hardware needs developer machines with at least two displays to be effective. A second display is incredibly inexpensive, even if you purchase one of those snazzy 3M multitouch displays. Without the second display, your developers will waste precious hours just in context switching between the Modern UI and the desktop during debugging sessions. You'll save time, money, and headaches by properly equipping your entire team.

Marking your methods with the `async` modifier causes the compiler to reconstruct the method, inserting possible suspend and resume points every place where you use the `await` operator. The compiler ensures all the state management is handled for you, so you don't need to really think about the fact that your method may have stopped running.

Using the `await` operator makes it very easy to support making asynchronous calls—all the magic required to support asynchronous operations is wrapped up in code you never see, much like compiler optimizations are. This approach works well for the majority of situations. But when you want to check the progress of an operation or cancel an operation in progress, you must use a slightly more complex approach.

Async and await under the covers

When you use `async` and `await` in your code, the compiler does a lot on your behalf. Any function that's marked using the `async` modifier is rewritten as a state machine. All local function state is stored in a structure.

The code that implements the actual functionality you wrote is also located in a method named `MoveNext` on that same structure. Local variables become fields in the structure. Your original function ends up being just a bit of a shell that initializes the state machine and passes control to it.

The implementation code starts with a `switch` statement that uses the current state as input. Depending on that state variable, the code branches to one of the asynchronous calls, a check to see if the calls have completed and if so, code to get the results, or the end of the function.

This state machine enables the function to be suspended and the state boxed onto the heap for reactivation once the asynchronous call returns.

Each completed step modifies the state. Each time the function is called, the opening `switch` statement branches the code so it picks up where it previously left off.

The end result is a lot of compiled code created on your behalf, too much to post in a book. If you're curious about the inner workings of the asynchronous state machine, you can see the generated MSIL (Microsoft Intermediate Language) and more in my blog post on this topic at http://bit.ly/asyncstatemachine.

16.2.2 *Long-form asynchronous operations*

The `async` and `await` keywords are conveniences but aren't required when working with asynchronous operations. You can still use the long-form approach with event handlers and callback functions.

The options depend on which interfaces the return type of the asynchronous function implements. Table 16.1 shows the different interfaces and describes their capabilities.

Table 16.1 Async support interfaces in WinRT. All asynchronous operations must return a type implementing at least one of these interfaces.

Interface	Description
`IAsyncAction`	The most basic async support interface beyond `IAsyncInfo`. This defines an asynchronous method that doesn't have a return type.
`IAsyncActionWithProgress<TProgress>`	An interface that supports progress reporting of a supplied type.
`IAsyncOperation<TResult>`	An interface that supports an asynchronous method that has a return value.
`IAsyncOperationWithProgress-<TResult,TProgress>`	An interface that supports an asynchronous method with both a return value and progress reporting.

In addition, the `IAsyncInfo` interface supports the asynchronous operations by providing `Cancel` and `Close` methods, as well as gets error and status information. The four interfaces include this capability because they inherit from `IAsyncInfo`.

Going back to our `SyndicationClient` example from the previous section, you can also use the long-form approach to make the call. In this case, the return type implements the most feature rich of the interfaces: `IAsyncOperationWithProgress`. That means when working with the `SyndicationClient`, you can get back a result but also get updates as the operation progresses. This is the signature:

```
public IAsyncOperationWithProgress<SyndicationFeed,RetrievalProgress>
                  RetrieveFeedAsync(System.Uri uri)
```

`SyndicationFeed` is the real return type—it's the data you want. If this function were implemented without asynchronous operation in mind, it might look like this:

```
public SyndicationFeed RetrieveFeedAsync(System.Uri uri)
```

But it's asynchronous, so the return type is a class implementing the interface. The next listing shows how to call the function using the long-form asynchronous pattern. Call this function from `OnNavigatedTo` just as you did in listing 16.1.

Listing 16.2 Downloading a syndication feed using the long-form approach

```
private void LoadFeedUsingLongForm()
{
  var client = new SyndicationClient();

  var uri = new Uri("http://feeds.feedburner.com/PeteBrown");
  var op = client.RetrieveFeedAsync(uri);

  op.Completed = (info, status) =>
    {
      switch (status)
      {
        case AsyncStatus.Canceled:
          Debug.WriteLine("Operation was cancelled");
          break;
        case AsyncStatus.Completed:
          Debug.WriteLine("Operation was completed");
          Debug.WriteLine(info.GetResults().Title.Text);
          break;
        case AsyncStatus.Error:
          Debug.WriteLine("Operation had an error:");
          Debug.WriteLine(info.ErrorCode.Message);
          break;
      }
    };
}
```

Completed delegate—a callback

Operation canceled

Operation complete

Operation error

Notice how `Completed` is not an event handler. Instead, it's simply a delegate property. For that reason, you use `=` rather than `+=` when adding your handler code. This subtle difference could trip you up if you don't recognize it right off the bat.

When you run this, the `Completed` delegate will be called, which will then write out the title of my blog in the debug window in the IDE. As before, you'll need to terminate the application from the IDE.

What about race conditions?

When you look at the code in listing 16.2, you may wonder about a possible race condition between executing the function and wiring up the `Completed` handler.

In regular .NET asynchronous code following the event approach, you need to wire up the event handlers before executing the operation. This is to ensure that the event handlers are available before the function needs them. For potentially fast async operations, this is especially important.

In the Windows Runtime, however, the operation begins execution as soon as you call the function. There's no way for you to wire up a handler in between declaration and execution. Could there be a race condition where the operation completes before the handler is available?

Luckily, the Windows Runtime team thought of this very situation and ensured it won't happen. The returned operation contains everything that's needed for asynchronous function management. If the function call completes before you wire up the handler, the operation is still around and is able to manage context and handlers. In that scenario, as soon as you make the handler available, the operation will call it.

So, you don't need to worry about race conditions when setting up your handlers. The team already did.

This same thought was put into `Task` for functions such as `ContinueWith`.

Knowing when the task is complete is essential. For some long-running tasks, you may want to get information about interim status.

16.2.3 Getting progress updates

Some especially long tasks, such as downloading a large file over a slow connection or processing a large amount of data, should report back progress information to the calling code. This becomes especially important in the tablet market where bandwidth varies greatly from location to location.

To support flexible progress reporting, the `IAsyncActionWithProgress` and `IAsyncOperationWithProgress` interfaces provide a way for an asynchronous function to provide status updates using whatever data format is appropriate for that function.

In this case, the reporting type is a `RetrievalProgress` structure that exposes two properties: `bytesRetrieved` and `totalBytesToRetrieve`. Using these two properties, you can figure out the percentage complete and percentage left. The following listing shows how.

Listing 16.3 Obtaining progress reports from the RSS feed downloader

```
private void LoadFeedUsingLongForm()
{
  var client = new SyndicationClient();

  var uri = new Uri("http://feeds.feedburner.com/PeteBrown");
  var op = client.RetrieveFeedAsync(uri);

  op.Completed = (info, status) =>
    {
      ...
    };

  op.Progress = (info, progress) =>
    {
      Debug.WriteLine(string.Format("{0}/{1} bytes {2:P}",
        progress.BytesRetrieved,
        progress.TotalBytesToRetrieve,
        (float)progress.BytesRetrieved /
               (float)progress.TotalBytesToRetrieve));
    };
}
```

Code in previous listing

Progress report

This code is identical to listing 16.2, with the addition of a `Progress` handler. During execution, at a frequency determined by the code you're calling, you'll receive progress updates through this handler.

When I ran this code, however, I got some pretty interesting results. The `totalBytesToRetrieve` was incorrect, reporting a meaningless number. If I changed the URI to http://10rem.net/blog/rss.aspx[5] instead, I got the correct results:

```
4096/205187 bytes 2.00 %
8192/205187 bytes 3.99 %
12288/205187 bytes 5.99 %
16384/205187 bytes 7.98 %
 [snip]
200704/205187 bytes 97.82 %
204800/205187 bytes 99.81 %
205187/205187 bytes 100.00 %
Operation was completed
Pete Brown's Blog (POKE 53280,0)
```

Not all services correctly report the total bytes. In this case, FeedBurner clearly does not. When performing your own download, you'll need to check that number and adjust your progress reporting appropriately. One way would be to show an indeterminate progress bar in the case of unreported total bytes.

What about long-running operations that you need to abort? How would you deal with that situation?

---

[5] My source RSS feed that I use to feed the far more scalable and reliable FeedBurner feed.

16.2.4 Canceling the operation

For especially long-running operations, such as downloading a file, you'll want to provide a way for the user to cancel. When I've had to do this in the past, I would provide a Cancel button, which would then set a flag. The code doing the long-running operation, typically in a loop or timer, would check this flag during each iteration. If the cancel flag was set, the operation would terminate and clean up after itself.

When working with asynchronous code in the Windows Runtime, the process is not all that different from the approach I described. But instead of setting a flag, you call the `Cancel` function on the operation. The executing code uses this to determine what steps to take.

The next listing is an update to listing 16.3, which will cancel after a set number of bytes have been downloaded.

Listing 16.4 Canceling the operation after 5,000 bytes have been downloaded

```
bool alreadyCancelled = false;
op.Progress = (info, progress) =>
  {
    Debug.WriteLine(string.Format("{0}/{1} bytes {2:P}",
        progress.BytesRetrieved,
        progress.TotalBytesToRetrieve,
        (float)progress.BytesRetrieved /
              (float)progress.TotalBytesToRetrieve));

    if (!alreadyCancelled)
    {
      if (progress.BytesRetrieved > 5000)
      {
        info.Cancel();
        alreadyCancelled = true;
      }
    }
  };
```

Prevent redundant Cancel calls

Prevent redundant Cancel calls

Cancel operation

This listing shows one approach to canceling. Typically, you'd use a button or other UI device to call the `Cancel` method, as stated previously.

One important thing to note from this listing is that when you run it, you may not see it actually cancel until the operation is complete. Remember, this is an asynchronous operation, and the function is responsible for deciding when it can cancel and how quickly to do so. For grins, if you change the 5000 to 0, you'll likely see it cancel right away.

Looking at these long-form approaches to asynchronous code may make you double-check to see if you accidentally hit 88 mph in a DeLorean while you were napping. It certainly may feel like we've gone backward, but with the addition of `async` and `await` now making this verbose approach optional, I can assure you that we're going in the right direction.

If you want the flexibility and power, choose the long/verbose version. If you don't need all that flexibility (which will likely be the majority of the time—do you really need progress reporting or cancellation when opening a file?), use the `async` and `await` keywords to make your life easier. Don't force yourself or your team to pick just one approach; use what works for the specific situation and keep it as simple as possible as often as possible.

Speaking of keeping things simple, the same `async` and `await` keywords also work with the .NET Task approach. The long-form version is a little different, however, so we'll cover that next.

16.3 Working with tasks

The Task Parallel Library was introduced with .NET 4.0. A small subset of it was added to Silverlight in Silverlight 5. .NET 4.5 saw much deeper integration of TPL and a reliance on it for asynchronous operations. The purpose of TPL was to provide a standardized way to write multithreaded and parallel code in .NET. As a part of that, it made creating asynchronous operations easier.

Once you venture outside the Windows Runtime and into the .NET libraries (System.* instead of Windows.*), you're far more likely to run across tasks than with the asynchronous interface-faced approach. The TPL approach to asynchronous functions is a .NET-specific approach, following common .NET patterns, which is why it wasn't rolled into WinRT. So, while what you learn here may not be applicable to JavaScript or even C++ (both of which have their own language-specific approaches), it's portable to .NET applications on the desktop and server.

In this section, we'll take a look at accomplishing asynchronous tasks using the `Task` type from the TPL. This section is not a complete (or even mostly complete) discourse on TPL, because we could write entire chapters (or books) on parallel programming. Instead, I'll focus on the practicalities of working with asynchronous APIs. As in the previous section, I'll start with showing basic task operations and then show how to get progress information. Next, we'll look at how to cancel tasks once they are in progress. This section will wrap up with a way to integrate WinRT and TPL to enable you to use a single asynchronous model across your application.

Using `async` and `await` is, from the consuming code's point of view, identical to that with WinRT. But as a baseline, let's take a quick look at using the `await` operator to download an HTML page from the web. The next listing shows the two places where you need to await an asynchronous result when using the `HttpClient`.

Listing 16.5 Asynchronous operation to download a web page using `Task` and `await`

```
private async void LoadHttpWithAwait()
{
  var client = new HttpClient();

  var uri = new Uri("http://10rem.net");
  var feed = await client.GetAsync(uri);

  var results = await feed.Content.ReadAsStringAsync();

  Debug.WriteLine(results);
}
```

async keyword

await first result

await stream processing

This example uses the .NET `HttpClient` class to download content from the web. The methods of that class use the `Task` type to manage asynchronous operations. But when looking at this code, you'll see that it's almost identical to the WinRT version. What's different? The main difference is that you needed to await two different operations in this method because of how stream processing is handled. That's not specific to .NET; it's specific to the function you're calling and the implementation of the `HttpClient` class.

One takeaway from this is that the `await` operator greatly simplifies your asynchronous code. Other than the `await` operator, it looks just like normal synchronous code. Compare that to the older event or callback-driven style approach you used in .NET and Silverlight, and you can see how helpful this new keyword is. There are no nested anonymous delegates nor anything else in the way.

Creating your own async methods

If I wanted to create my own `async` version of the first part of this method, it might look something like this:

```
private Task<HttpResponseMessage> LoadSiteAsync()
{
  var client = new HttpClient();
  return client.GetAsync(new Uri("http://10rem.net"));
}
```

The code simply returns the `Task` provided from the `GetAsync` method. Of course, this is a really simple implementation of an `async` method. If you want to create something more complex, using your own processing and logic, it might look something like this:

```
private Task<string> DoSomethingAsync()
{
  return Task.Run<string>(()=>
    {
      return "I am an async string";
    });
}
```

You can do whatever you want inside the `Run` block, of course. The `Task.StartNew` method provides additional parameters to help control the behavior of the task. `Task.Run` is simply a convenience method covering the majority of common usages.

With the baseline set, let's look at the long-form version of this call.

16.3.1 Basic task operations

Just as was the case with `IAsync*` operations in WinRT, the `Task`-based approach has a longer form that provides more functionality over the simpler `await` approach. Like the WinRT approach, you'll use this from time to time when you need additional hooks into the task. Unlike the WinRT approach, you don't have progress reporting. But you can do cancellation, which we'll look at in a moment.

First, you need to level set what a long-form `Task` implementation looks like. It's not complex, but it is different from what we had with WinRT. The following listing shows the same functionality from listing 16.5 implemented using the verbose syntax.

Listing 16.6 Using the more verbose form without `await`

```
private void LoadHttpUsingLongForm()
{
  var client = new HttpClient();

  var uri = new Uri("http://10rem.net");

  client.GetAsync(uri).ContinueWith((t) =>
    {
      t.Result.Content.ReadAsStringAsync().ContinueWith((t2) =>
        {
          Debug.WriteLine(t2.Result);
        });

    });
}
```

First async operation

Second async operation

This listing shows the two asynchronous operations written out using the long-form approach with delegates that are executed on completion of the `Task`. This looks quite different from the WinRT version because the .NET version makes use of extension methods and lambda expressions—constructs not available in every other language.

In this code, `GetAsync` returns a value of type `Task<HttpResponseMessage>`. The `HttpResponseMessage` class includes a number of properties like `Headers`, `StatusCode`, and what we really want: `Content` that is of type `HttpContent`. That type, in turn, includes several methods that can be used to get the content data. These methods are all asynchronous using the `Task` type and pattern. Because those methods, including `ReadAsStringAsync`, are asynchronous, I need to spin up another task as shown here or use the `await` keyword.

Listing 16.6 can be written in a more compact form should you desire. For clarity of when things execute, I included the braces. The slightly shorter form of the asynchronous operation part of the function could look like this:

```
client.GetAsync(uri).ContinueWith((t) =>
    t.Result.Content.ReadAsStringAsync().ContinueWith((t2) =>
        Debug.WriteLine(t2.Result)));
```

Even the long form for using tasks is surprisingly succinct, especially when you leave out the braces. But unlike the WinRT `IAsync*WithProgress` interfaces, the `Task` class doesn't include any standardized built-in support for incremental updates or progress reporting,[6] so we'll skip right to canceling the task.

---

[6] There's `System.IProgress<T>` but that is really only used for converting from WinRT to TPL tasks.

16.3.2 Canceling the task

Like the Windows Runtime asynchronous approach, the `Task` approach provides a way to cooperatively cancel an operation. I say *cooperatively*, because just as with WinRT, both the asynchronous code and the calling code must agree to cancel the `Task`. On the calling side, this is done using a `CancellationTokenSource`, which contains a `CancellationToken` structure. That structure is passed in as a parameter to the asynchronous function.

The following listing shows how to create an instance of the `CancellationTokenSource` and how to provide it to the `GetAsync` function.

Listing 16.7 Canceling an in-progress `Task`

```
private void LoadHttpUsingLongFormWithCancel()
{
  var client = new HttpClient();

  var uri = new Uri("http://10rem.net");

  var cancelTokenSource = new CancellationTokenSource();
  var cancelToken = cancelTokenSource.Token;

  client.GetAsync(uri, cancelToken).ContinueWith((t) =>
    {
      if (t.IsCanceled)
        Debug.WriteLine("Download was cancelled");
      else
        t.Result.Content.ReadAsStringAsync().ContinueWith((t2) =>
             Debug.WriteLine(t2.Result));
    });

  cancelTokenSource.CancelAfter(5);
}
```

Create listener

Provide token

Check for cancellation

Force cancellation after 5 ms

As described, this code handles the creation of the `CancellationTokenSource`. Once the token from that source is passed in to `GetAsync`, it's the responsibility of the `GetAsync` code to check if a cancel has been requested. In this case, you cancel after just 5 ms of running, as shown by the last line of code. The cancellation itself is performed on the token source, which then notifies the asynchronous code via the token. The same token can be passed into any number of asynchronous calls; you could have passed it into `ReadAsStringAsync` as well.

Inside the `ContinueWith` delegate, you first check to see if the task was canceled—`ContinueWith` executes this code regardless of how the `Task` completed. If canceled, you display a message indicating as much. Otherwise, you display the download results just as in the previous listings.

By now you've probably formed an opinion more in favor of `IAsync*` or `Task`. You may think you're stuck using both of them in your apps, doubling the things any developer maintaining your code needs to know. Not to worry, the product teams thought of you and came up with nifty ways to convert between the two.

16.3.3 Converting between WinRT IAsync and Tasks*

I like both the `IAsync*` and `Task` approaches, but in any given application, it would be ill-advised to mix and match them. There's no technical reason why you should pick one or the other, but you want to make the code easier for other developers to understand. That means you likely want to use one or the other.

"But wait—if I standardize on a single approach, does that limit the APIs I can use?" you may ask. Happily, the answer is no. The `WindowsRuntimeSystemExtensions` class in the `System` namespace in the `System.Runtime.WindowsRuntime` assembly includes a number of extensions that help convert between the different approaches. Because this is already referenced by your application, the extension methods automatically appear on the correct types.

CONVERTING FROM TASKS TO IASYNC*

If you decide you like the Windows Runtime interface-based pattern for asynchronous operations but want to use a .NET class that implements the `Task`-based pattern, you'll need to perform a simple conversion.

`Task`s, because they have no built-in progress reporting, can be converted only to `IAsyncOperation` or `IAsyncAction`. This is accomplished by using the `AsAsyncOperation` and `AsAsyncAction` extension methods, respectively.

The next listing shows how to use this conversion with the `Task`-based `HttpClient` `GetAsync` method.

Listing 16.8 An example of converting a `Task` to `IAsyncOperation`

```
private void ConvertTaskToWinRT()
{
  var client = new HttpClient();

  var uri = new Uri("http://10rem.net");

  var op = client.GetAsync(uri)
             .AsAsyncOperation<HttpResponseMessage>();

  op.Completed = (info, status) =>
    {
      switch (status)
      {
        case AsyncStatus.Canceled:
          Debug.WriteLine("Operation was cancelled");
          break;
        case AsyncStatus.Completed:
          Debug.WriteLine("Operation was completed");
          break;
        case AsyncStatus.Error:
          Debug.WriteLine("Operation had an error:");
          Debug.WriteLine(info.ErrorCode.Message);
          break;
      }
    };
}
```

Conversion

This simple conversion enables you to convert from `Task` to `IAsync*`. The underlying `Task` instance is still valid and usable, but I don't recommend using it, because mixing both in the same operation is a recipe for confusion.

Of course, you can also convert in the opposite direction.

CONVERTING IASYNC* TO TASK

Many .NET developers will be, at least initially, more comfortable with `Task`-based approaches. In addition, there's already a lot of code out there that relies on the `Task` object. You may even have some in your own apps. To make it easier to reuse that code, you may want to convert `IAsync*` usage to `Task`.

You can convert any of the four `IAsync*` interface approaches to an equivalent `Task`-based approach. Significantly, you can even retain the progress reporting through the purpose-built `IProgress` interface.

The following listing shows how to use the feed downloader with a `Task`-based asynchronous model.

Listing 16.9 Converting `IAsyncOperation` to `Task`

```
private void ConvertWinRTToTask()
{
  var client = new SyndicationClient();

  var uri = new Uri("http://feeds.feedburner.com/PeteBrown");

  var task = client.RetrieveFeedAsync(uri)
              .AsTask<SyndicationFeed, RetrievalProgress>();

  task.ContinueWith((t) =>
    {
      if (t.IsCanceled)
        Debug.WriteLine("Feed retrieval was cancelled");
      else
        Debug.WriteLine(t.Result.Title);
    });
}
```

Conversion

As was the case when converting from `Task` to `IAsync*`, you can continue to use the original interfaces, but I caution against it purely for reasons of sanity.

Being able to convert between the two asynchronous models gives you the choice to use whatever you're most comfortable with. It also helps make the marriage of .NET and WinRT seem more like a single API as opposed to two different sets of libraries.

The `Task`-based approach is appealing both for historical and syntactical reasons. Some people will just naturally prefer the lambda-style syntax and the synergy with the rest of the TPL. Others will prefer the tried-and-true interface-based approach favored by WinRT. Either way, you get the UI responsiveness that asynchronous operations help to ensure.

16.4 Summary

The Windows Runtime includes asynchronous methods all over. Additionally, the .NET libraries used for writing C# Modern Windows Store apps also include asynchronous code. But those libraries use the `Task` class rather than the WinRT interface-based asynchronous approach. This helps keep .NET consistent with the desktop version and also with the previous version of .NET. Gladly, you can use `async` and `await` just as easily with a .NET `Task` as you can with a WinRT `IAsync*` interface. Plus, if you really want to keep your code consistent, you can easily convert between the two approaches using included extension methods.

Asynchronous code used to be something we dreaded in application development. With both `Task` and `IAsync*`, and the new `async` and `await` keywords, combined with the appropriate use of asynchronous operations throughout the libraries, I feel, for the first time, as if we really have asynchronous patterns done right. Async everywhere? I'm cool with that.

In the next chapter, we'll look at a very practical use of asynchronous code: networking operations.

17

Networking with SOAP and RESTful services

This chapter covers

- Networking basics
- Working with SOAP services
- Using REST services, shared models, and MVVM
- Serializing JSON and XML

It's rare to find an app that doesn't use resources out on a network. We live in a world of connected applications. Even apps as benign as games often phone home with metrics and usage data, high scores, update checks, ad metrics, and much more. For an application platform to be considered viable, it must have first-class networking support.

In this chapter, we'll first take a quick look at the networking APIs available for use in WinRT and .NET 4.5. From there, we'll dive right into learning about integrating with SOAP services. Even though SOAP services are on the decline, you'll still run into them, and for some types of procedural operations, they're the best choice.

Once we clean up with SOAP,[1] we'll move on to a network-friendly implementation of MVVM, this time without using any MVVM toolkit. We'll then use that MVVM implementation to structure the code we'll use when integrating with an ASP.NET Web API RESTful service. That service may return XML or JSON, so we'll also dive into how to manage serialization and deserialization with those two formats. But, before we can get going with services, let's look at the basics, starting with downloading a file and setting up the site you'll use throughout the rest of the chapter.

---

[1] Oh yes I did!

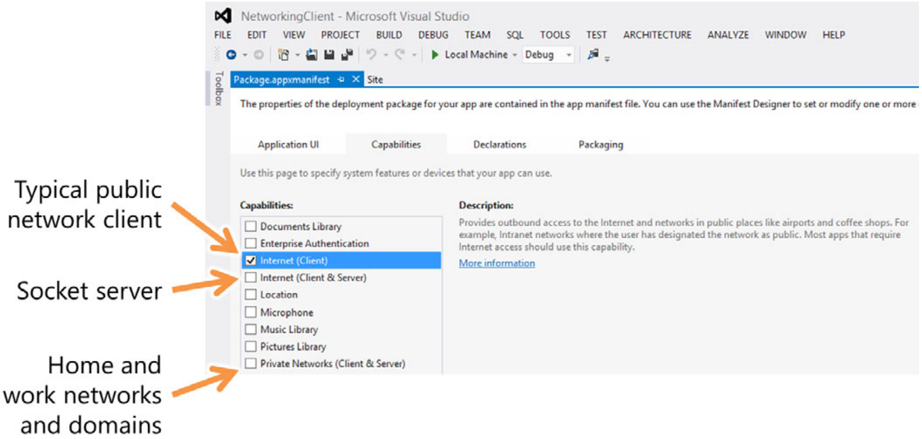
17.1 Networking basics

Networking is core both to the Windows Runtime and to .NET 4.5. In the Windows Runtime, you'll find classes for background transfer, checking connectivity, working with RSS and ATOM feeds, and much more. Many of those we'll look at in this chapter and the next. In .NET 4.5, you'll find familiar yet updated versions of `HttpClient` and `(Http)WebRequest`.

The classes may look familiar, but they're not identical to what you may have experience with. In particular, if you're familiar with Silverlight, you may assume we have the same client access policy limitations in Windows 8 apps. I'm happy to say we do not—apps that have the Internet Client capability (which they all request by default) are able to make HTTP calls out to any internet site without any special policy files. (I'll pause here for applause.)

To start the project, create a new Windows Store C# XAML application using the Blank App (XAML) template. I named mine NetworkingClient. You'll use this app throughout the remainder of this chapter.

To check to see if you have requested the Internet Client capability, double-click the Package.appxmanifest file and then click the Capabilities tab. Check to make sure the Internet (Client) item is checked, as shown in figure 17.1.


NetworkingClient - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Local Machine
Debug
Toolbox
Package.appxmanifest
Site
Application UI
Capabilities
Declarations
Packaging
More information
Typical public network client
Socket server
Home and work networks and domains

The properties of the deployment package for your app are contained in the app manifest file. You can use the Manifest Designer to set or modify one or more

Use this page to specify system features or devices that your app can use.

Capabilities:

- ☐ Documents Library
- ☐ Enterprise Authentication
- ☑ Internet (Client)
- ☐ Internet (Client & Server)
- ☐ Location
- ☐ Microphone
- ☐ Music Library
- ☐ Pictures Library
- ☐ Private Networks (Client & Server)

Description:

Provides outbound access to the Internet and networks in public places like airports and coffee shops. For example, Intranet networks where the user has designated the network as public. Most apps that require Internet access should use this capability.

Figure 17.1 The Internet Client capability shown in the designer for the Package.appxmanifest file. Also note the other two networking capabilities: One is for servers (and clients) on public networks, and the other is for servers and clients on private networks and domains.

Although there are many other higher-level HTTP networking classes and lower-level raw networking classes as mentioned earlier, when it comes to basic HTTP networking, there are two main classes you'll run across:

- `HttpClient`—Use the `HttpClient` class when you don't need more than the ability to specify one of the standard four (GET, PUT, POST, DELETE) HTTP verbs, and you don't need to do anything fancy with headers or security.
- `HttpWebRequest`—Use this class when you need finer control over headers, verbs, and more. If you want to use the relatively new PATCH verb, this is the class you'll need to use.

We'll take a look at those two classes in this chapter, as well as the code that builds on them, such as that generated when you add a service reference.

To effectively test networking, you need an endpoint. In this section, you'll set up the solution and web project that will be used for the file download as well as the SOAP and REST examples. We'll also look at one of the easiest-to-use networking classes, `HttpClient`, and download a file using its methods.

17.1.1 Solution setup

Before we get into how to use networking, you'll need to create an endpoint (or several) for your calls. In order to make these examples train-friendly[2] and reduce outside dependencies, the endpoints for each of the examples in this chapter will be created in an in-solution web project.

Networking on the cheap with Visual Studio Express 2012

Those of you using the free edition of Visual Studio Express for Windows 8 for building Metro-style applications may wonder how you can create web server projects. In the networking section of this book, I generally assume you have Visual Studio Professional 2012 or better, as people writing SOAP services and home-grown RESTful services usually do.

Most of the code here can be easily adapted to work with APIs for your favorite sites. Twitter, Flickr, and many others offer web-friendly APIs. The concepts you learn here will certainly apply to those sites.

Nevertheless, for developers without access to the higher-level Visual Studio SKUs, but who want to use these examples, there are two good options:

- *Use existing Visual Studio 2010 installations.*
  If you already have Visual Studio 2010, you can run it side by side with Visual Studio 2012. You won't have access to anything .NET 4.5-specific in the VS2010 project, but you'll at least be able to set up web services or sites that you can use.

---

[2] I can get Wi-Fi reliably at 35,000 feet in the air, but for some reason, I can never get it to work on the train. First-world problem, I know.

(continued)

- *Install Visual Studio 2012 Express for Web.*
  You can download and install the free web edition of Visual Studio 2012. This will give you access to .NET 4.5 features, and you can run it side by side with Visual Studio Express for Windows 8.

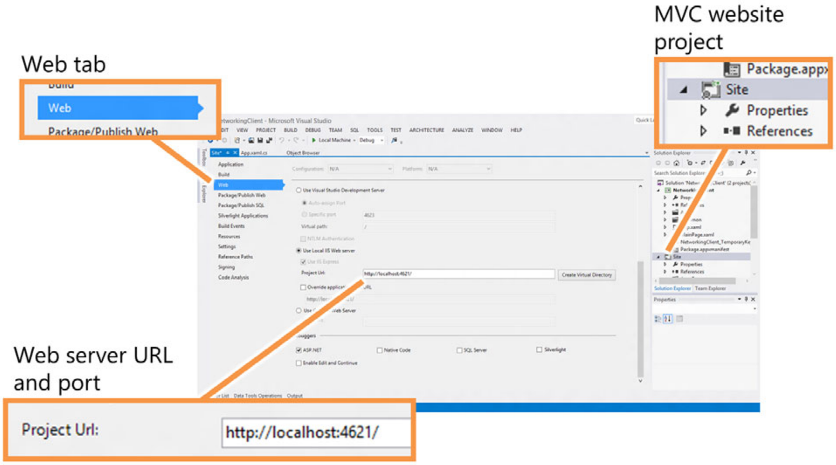
Either of these alternatives will require you to write the server-side code in a separate IDE and start it running before you run your Windows app or try to add a service reference. You'll not be able to use Discover to add a service reference for SOAP services when running this way; instead, you'll have to enter the URL directly in the Add Service Reference dialog.

This is the way we used to debug applications before Visual Studio started supporting running the web app automatically. It's not the most convenient approach to debugging, but if you want something free, sometimes you need to give up a small amount of convenience.

That said, even if you have the full Ultimate edition of Visual Studio 2012, you can use this same approach. If debugging fails with all the projects in the same solution, or if keeping the site alive between debugging sessions (to retain cache, for example) is important, having two separate IDE instances running can really be a life saver.

Add to the solution an ASP.NET MVC 4 web project. You could likely run the examples in this chapter using a web forms application if you prefer, because there's nothing here that's particularly MVC-specific, but I'm going to focus on MVC. The RESTful work you'll do in this chapter pairs well with MVC but doesn't require it.

I named the new MVC project quite simply Site and used the Empty MVC project template with Razor support when prompted to pick a template and view engine. The



Project Url: http://localhost:4621/

Figure 17.2 Settings on the web tab of the website project. Take note of the URL shown on your own installation because you'll need to know it, including the port number, in this section. Your port number will likely be different from mine.

specifics aren't that important, as long as you can identify the URL to the file and services you'll add.

Once the project has been created, right-click the web project and select Properties to pull up the Property pages. Once the property pages are open, select the Web tab and take note of the URL for the web server. If you don't like that URL for some reason, now is the time to change it, because you'll rely on it for the various examples in this chapter. Figure 17.2 shows the settings page on my machine. You don't need to match my URL and port; you just need to stay consistent with your own.

Finally, create a new HTML file in the root of the MVC site project. Name it something you'll remember. I used fan.html. The actual contents of the file aren't important; the following listing shows mine.

Listing 17.1 The fan.html file you'll use for the file-downloading example

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>The Greatest HTML File on Earth</title>
</head>
<body>
I work equally well in all browsers. How cool is that!
</body>
</html>
```

With the client project, web project, and HTML file in place, you're now all set to try out a little networking. Your first task will be to download a file.

17.1.2 Downloading a file with HttpClient

One of the easiest and most fundamental practical networking operations is downloading a file from a web server. When it comes down to it, most operations performed across HTTP are really just downloading a file, whether it's an actual file on the filesystem or some results dynamically generated from a web service. In this example, you'll download the HTML file you created in the site project earlier and then display its contents in the debug window.

Do you remember the URL for your web project? If not, you may find it in the project Property pages, in the properties pane when you select the project file, and also in the address bar if you run the web project or right-click and view the HTML file in the browser. If you put the file in the root, the full URL will be the web server address plus the filename. On my machine, that's

```
http://localhost:4621/fan.html
```

Open the MainPage.xaml.cs file and find the `OnNavigatedTo` method. Mark that method using the `async` modifier, as you learned in chapter 16. Next, you'll use the `await` operator to call the asynchronous `GetAsync` function to download the file. The full code to download and display the file contents is shown in the next listing.

Listing 17.2 Downloading a file and displaying its contents

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    var uri = new Uri("http://localhost:4621/fan.html");
    var client = new HttpClient();

    var opContent = (await client.GetAsync(uri)).Content;

    string fileContents = await opContent.ReadAsStringAsync();

    Debug.WriteLine(fileContents);
}
```

async modifier

HttpClient

Get operation

Actual results

This example shows two different asynchronous methods. The first performs the `Get` operation. The second reads the stream of data and converts it to a `string`. Because you use the `await` operator, covered in chapter 16, the code remains simple despite the number of asynchronous operations it contains. Don't forget to add the `async` modifier to the event handler declaration—that's an easy one to miss.

You'll need to add `using` statements for the `System.Diagnostics` and `System.Net.Http` namespaces. Make sure the Windows Store app project is the startup project. Once you do that, compile and run. Give the application a moment after the default loading image disappears and then Alt-Tab (if you have only a single monitor) to the IDE and check out the debug window. You should see the contents of the HTML file.

If the only network operation you ever learn is the one required to download a file, you can still go quite far. It's the most essential HTTP networking method and the easiest to understand.

For the remainder of this chapter, we'll work with services. Although the code will, at its heart, be similar to the simple file download, it will have the added complexity of mapping that downloaded data to some sort of class or structure. Before we tackle that, let's look at a best practice: sharing your model between client and server.

17.2 Sharing your model

In the previous section, you downloaded just a flat file. Essentially, the model for that result would be a string. But it's far more common to download something that can be represented as a class, such as a customer, or a message, or a Tweet. Because you'll be working with services for the remainder of this chapter, now is a good time to introduce a best practice for sharing those classes (collectively called *the model* or *entities*) between the client and server.

There are two approaches to defining and sharing your model:

- *Define the classes directly in the service project.*

  Include those classes as part of the service definition when using services that generate proxy classes. In this case, you'd allow the service reference code generator to automatically generate the client-side versions of the model classes.

- *Define the classes externally.*

  In this case, you'd create them in separate projects and share them between the server project and the client project. This is more flexible and works better, because you don't get hung up by any type limitations in the code generator. It's also not magic-generated code—you have complete control over the definitions.

I stopped using the first approach ages ago, even though it's the one most easily done with the existing tooling. The second approach is the one many people turn to when they've exhausted the capabilities of the first approach. Rather than start one way and end up refactoring to another, I prefer to start with the second approach on every project.

Why is the second approach better?

- *It facilitates conditional compilation.*

  In some instances, it makes sense to use conditional compilation to extend the shared classes for a specific platform. For example, you may want to add `INotifyPropertyChanged` support in one project but not the other.
- *It facilitates platform-specific extension.*

  If you define the classes as `partial`, you can extend them with platform-specific code. For example, a client class (not a model class, specifically) may have code that performs a lookup using a service, whereas server-side code may perform that lookup using cached data or a local database.
- *You can use additional types.*

  Especially in older versions of Visual Studio, the code generator didn't handle all types correctly—you were limited to a subset of acceptable .NET types, even if the serialization code understood them correctly.
- *It works when there's no proxy generator.*

  Some types of services, particularly RESTful services at the moment, do not support generation of client-side proxies through an Add Service Reference feature. In those cases, you really do need to take an approach like this to share entities between a client and server project, assuming you own both.
- *The approach works for more than just services.*

  This is exactly the same approach I use when sharing implementation code between Silverlight and Windows Store apps, Silverlight and WPF, Windows Store apps and Phone, and so on.

Hopefully I've convinced you of the benefits of this approach by now. If not, just humor me for the rest of the chapter.

In the rest of this section, you'll set up the project structure required to share strongly typed model projects between ASP.NET and the Windows Store app. You'll first create the source class library and then look into how to link files. Once you understand file linking and the relationship between the projects, you'll create the linked class library.

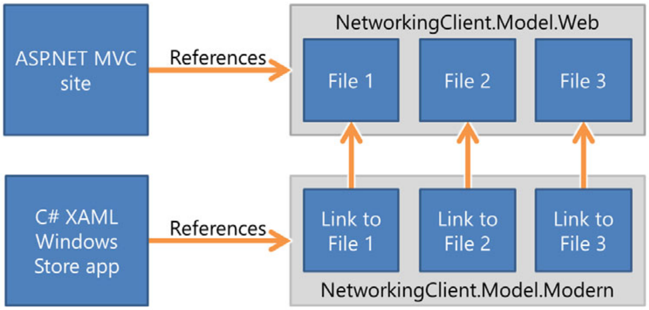

ASP.NET MVC site
References
NetworkingClient.Model.Web
File 1
File 2
File 3
C# XAML Windows Store app
References
Link to File 1
Link to File 2
Link to File 3
NetworkingClient.Model.Modern

Figure 17.3 Linked files in a Visual Studio project. This is how to handle dual-targeting in most cases, especially when sharing `Model` objects between services and the client.

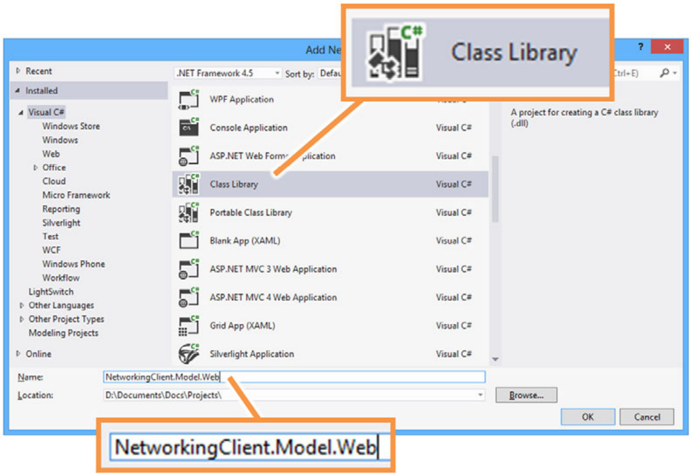
17.2.1 Create the source class library

When sharing source code, a best practice is to designate one project as the *master* or *source* and any others as the copies. You don't *have* to do this, you could share files directly between the client project and the web server project, but many have found that approach to be messy and one which creates too large a project dependency.

In our solution, the project NetworkingClient.Model.Web will contain the source files. Another project, which targets Modern Style Windows Store apps, will be named NetworkingClient.Model.Modern and will contain the links to the source files. Figure 17.3 illustrates the relationship.

If you're using the free edition of Visual Studio Express 2012 for Windows, you won't be able to create the MVC site or the class library in the same solution. The techniques for using multiple instances of Visual Studio (see the sidebar earlier in this chapter) work here as well.

As mentioned, there are two different projects involved in this code sharing. You'll have only a single source class, the `InstantMessage` model class, but retain the multiproject approach. Create a regular .NET 4.5 class library and name it `Networking-Client.Model.Web`, as shown in figure 17.4.


Add Ne
?
×
Recent
Installed
Visual C#
Windows Store
Windows
Web
Office
Cloud
Micro Framework
Reporting
Silverlight
Test
WCF
Windows Phone
Workflow
LightSwitch
Other Languages
Other Project Types
Modeling Projects
Online
.NET Framework 4.5
Sort by: Defau
Ctrl+E)
WPF Application
Console Application Visual C#
ASP.NET Web Form plication Visual C#
Class Library Visual C#
Portable Class Library Visual C#
Blank App (XAML) Visual C#
ASP.NET MVC 3 Web Application Visual C#
ASP.NET MVC 4 Web Application Visual C#
Grid App (XAML) Visual C#
Silverlight Application Visual C#
Class Library
Browse...
OK
Cancel

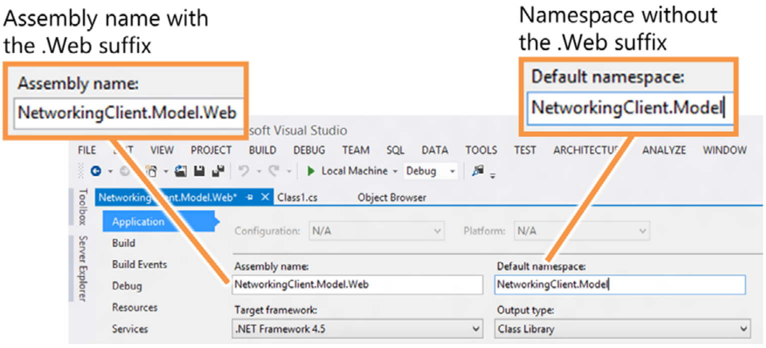
A project for creating a C# class library (.dll)

Name: NetworkingClient.Model.Web

Location: D:\Documents\Docs\Projects\

NetworkingClient.Model.Web

Figure 17.4 Creating the model project as a full .NET 4.5 class library (DLL) project



Assembly name with the .Web suffix

Namespace without the .Web suffix

Assembly name: NetworkingClient.Model.Web

Default namespace: NetworkingClient.Model

Configuration: N/A

Platform: N/A

Target framework: .NET Framework 4.5

Output type: Class Library

soft Visual Studio
FILE
VIEW
PROJECT
BUILD
DEBUG
TEAM
SQL
DATA
TOOLS
TEST
ARCHITECTU
ANALYZE
WINDOW
Local Machine
Debug
Toolbox
Server Explorer
Networking nt.Model.Web*
Class1.cs
Object Browser
Application
Build
Build Events
Debug
Resources
Services

Figure 17.5 To facilitate code sharing later, the model project has the `.Web` suffix removed from the namespace but not from the project or assembly name.

Once the project has been added to the solution, open its property pages and remove `.Web` from the default namespace in the Application tab. This is necessary to help ensure the namespaces are the same generic namespace in each project. Without that change, because of the shared source files, you'd have a `.Modern` namespace in use in the web project or a `.Web` namespace in use in the client project. The assembly names are kept separate, however, to avoid deployment and reference confusion. Figure 17.5 shows the changes.

Back in the newly created model project in the Solution Explorer, delete the default Class1.cs file[3] and add a new class file named InstantMessage.cs. The following listing shows the code for that class. Note that the namespace is simply `NetworkingClient.Model`.

Listing 17.3 The `Message` class in the model project

```
using System;

namespace NetworkingClient.Model
{
  public class InstantMessage
  {
    public int Id { get; set; }
    public string From { get; set; }
    public string To { get; set; }
    public DateTime Timestamp { get; set; }
    public string Text { get; set; }
  }
}
```

Correct namespace

Properties

In this listing, all the properties are created with get/set autoproperty syntax. If you defined these just as public fields, you would be unable to bind to them from the UI. Both this detail and the namespace will facilitate sharing this code between the client and the server.

---

[3] I've never been given a satisfactory reason as to why we insist on keeping that file in the template. I know you can use refactoring and have a file rename all references, but then the namespace could still be wrong.

Now that you have the model project in place, it's time to add a reference to it from the web project. Right-click the website project, select Add Reference, and pick the NetworkingClient.Model.Web project. It's easy to close that dialog without actually adding a reference—make sure the reference is added by looking in the references folder of the Site project.

Portable Class Libraries

Portable Class Libraries (PCL) have come a long way over the past year or two. You'll find that they can do a lot of what linked files can do, without the overhead of creating multiple projects. For most code-reuse scenarios, PCL will work just fine.

I still prefer the linked-file approach because it provides maximum flexibility in exchange for simply remembering to add files as links. But I don't have to share my source code with a lot of teammates actively working on it, forgetting to add files as links.

Use the approach that works best for you and your team. Definitely investigate PCL, especially now that it supports all the major Microsoft platforms. In many cases, starting with PCL is the correct approach, because it's easy to switch to separate projects without rewriting code or otherwise changing much solution plumbing.

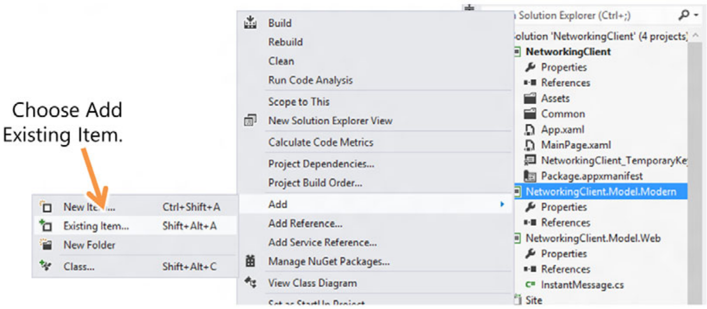
For more information on PCL, see http://bit.ly/NetPCL.

17.2.2 *Create the Modern app–compatible class library*

The next step is to create the client, or Modern app, class library. I know this may seem like a lot of work just to get a single class ready for use from a web service. But remember, we're working to follow best practices. On real applications, this little bit of extra effort will really pay off.

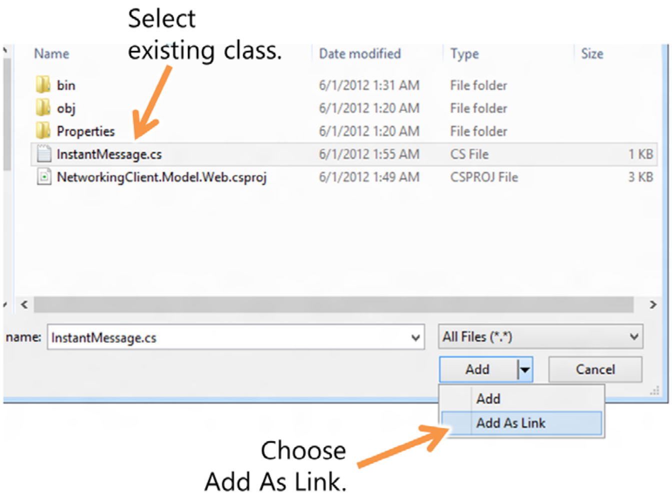
Referring to the .Web model project, follow the same steps to create the project, but use the Class Library (Windows Store apps) project template. The project and assembly filename should both be set to NetworkingClient.Model.Modern. As before, the namespace needs to be `NetworkingClient.Model`. Finally, remove the Class1.cs just as you did before, but don't add a new class to replace it.

Next, right-click the Windows Store class library and choose the option to add an existing item, as shown in figure 17.6.


New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Class... Shift+Alt+C
Build
Rebuild
Clean
Run Code Analysis
Scope to This
New Solution Explorer View
Calculate Code Metrics
Project Dependencies...
Project Build Order...
Add
Add Reference...
Add Service Reference...
Manage NuGet Packages...
View Class Diagram
Solution Explorer (Ctrl+;)
olution 'NetworkingClient' (4 projects)
NetworkingClient
Properties
References
Assets
Common
App.xaml
MainPage.xaml
NetworkingClient_TemporaryKe
Package.appxmanifest
NetworkingClient.Model.Modern
Properties
References
NetworkingClient.Model.Web
Properties
References
InstantMessage.cs
Site

Choose Add Existing Item.

Figure 17.6 Adding an existing item to the Modern Windows Store app model project



Name	Date modified	Type	Size
bin	6/1/2012 1:31 AM	File folder	
obj	6/1/2012 1:20 AM	File folder	
Properties	6/1/2012 1:20 AM	File folder	
InstantMessage.cs	6/1/2012 1:55 AM	CS File	1 KB
NetworkingClient.Model.Web.csproj	6/1/2012 1:49 AM	CSPROJ File	3 KB

name: InstantMessage.cs

Figure 17.7 Adding the existing `InstantMessage` class as a link

When you are presented with the File dialog to pick the file you want to add, don't be too quick about clicking Add. Instead, find the InstantMessage.cs source file from the NetworkingClient.Model.Web project, select that file, and then click the small drop-down button on the Add button and select Add As Link, as shown in figure 17.7.

Add As Link doesn't copy your file to the new project. Instead, it creates a pointer to that project. The same physical source file will be used in both projects, but it will be compiled into the binaries using the project settings specific to each project. If you look closely at the files in the Solution Explorer, you'll even see a little shortcut symbol on the file in the Windows Store project. If you don't, remove the file and readd the link.

As before, add a reference to the class library. This time, add it from the Windows Store client app to the Windows Store class library.

You now have code sharing between the full .NET framework (.Web project) and the Modern Windows Store subset of the .NET framework (.Modern project). That means you can share the same class definition between both projects. The next step is to create the SOAP service and then consume it from the Windows Store project.

17.3 Consuming SOAP services

SOAP, which originally stood for Simple Object Access Protocol but is no longer considered an acronym,[4] is an older, and somewhat heavier, standard for accessing data and functions across HTTP. It's incorrect to say "a" standard, because SOAP to most people means a lot more than the base definition. There are any number of WS-* protocols that differ in implementation from server to server, just adding to developer burden when using SOAP services.

---

[4] Despite no longer being considered an acronym, it's still written in all caps. Clearly this is the inspiration for the menu system in Visual Studio.

Because of the complexity of manually creating a SOAP envelope and the reliance on well-formed and valid XML, SOAP services are rarely seen outside the enterprise in modern code. Nevertheless, they're used and remain important to many inside organizations, so I'll cover the basics here.

In this section you'll create a WCF web service in the ASP.NET project. For simplicity, the web service will use our old friend the Silverlight Enabled Web Service template, which creates a really easy-to-understand SOAP service. Once you've created the service, you'll add a reference to it and then use the service from the Windows Store app.

17.3.1 Creating the service

First, create a Silverlight-enabled WCF service. This is a template you'll find under the Silverlight folder in the new project item dialog. No, I'm not sneaking some Silverlight into this solution. Instead, that template is the simplest WCF service template you can use with just about any client.

> **TIP** You could also use a standard ASP.NET .asmx web service, but my code examples will be for the WCF service implementation.

Create the service in the Services folder of the MVC project (if the folder isn't there, create it off of the root of the project), and name the service MessageService.svc. The service has a single method named `GetMessages`, which returns a `List<T>` of `InstantMessage` objects. The next listing shows the method implemented in a Silverlight-enabled WCF service.

Listing 17.4 The `GetMessages` method of the MessageService web service

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Activation;
using NetworkingClient.Model;                          <- Model project

namespace Site.Services
{
  [ServiceContract(Namespace = "")]
  [AspNetCompatibilityRequirements(RequirementsMode =
     AspNetCompatibilityRequirementsMode.Allowed)]
  public class MessageService
  {
    [OperationContract]
    public List<InstantMessage> GetMessages()                <- Service method
    {
      var messages = new List<InstantMessage>()              <- Hardcoded data
      {
        new InstantMessage() { From="Nappa", To="Vegeta",
          Text="Vegeta, what does the scouter say about his power level?",
          Timestamp=DateTime.Parse("6/1/2012 2:57pm"), Id=1 },
        new InstantMessage() { From="Vegeta", To="Nappa",
          Text="It's over 9000!!",
          Timestamp=DateTime.Parse("6/1/2012 2:58pm"), Id=2 },
    new InstantMessage() { From="Nappa", To="Vegeta",
     Text="9000!?",
     Timestamp=DateTime.Parse("6/1/2012 2:59pm"), Id=3 },
    new InstantMessage() { From="Nappa", To="Vegeta",
     Text="There's no way that can be right! It can't!",
     Timestamp=DateTime.Parse("6/1/2012 3:00pm"), Id=4 },
   };

   return messages;
  }
 }
}
```

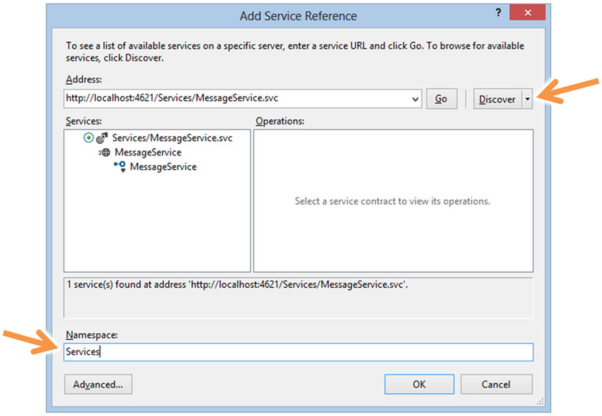
This service method uses the `InstantMessage` type defined in the model project. For simplicity, you return a hardcoded set of four messages in a `List` of messages. Typical services would pull from a database, cache, or other source, but it makes no material difference for this example.

17.3.2 Referencing and using the service

Consuming a web service in .NET 4.5 and client apps couldn't be simpler. All you need to do is add a service reference and then call the service method using the async pattern described in chapter 16.

Double-check to make sure your client app has a reference to the model project. If you forgot to add the reference to the model project before adding the service reference, the client-side proxy will include a copy of the definition for the `InstantMessage` class; you don't want that.

Build the solution, or at least build the web project. You'll probably get an error in the dialog if you fail to do this. Then, right-click the NetworkingClient app project and select Add Service Reference. You'll be presented with the dialog shown in figure 17.8.



To see a list of available services on a specific server, enter a service URL and click Go. To browse for available services, click Discover.

Address: http://localhost:4621/Services/MessageService.svc

Services:

Operations:

Select a service contract to view its operations.

1 service(s) found at address 'http://localhost:4621/Services/MessageService.svc'.

Namespace: Services

Click to discover services in solution.

Set namespace for generated client proxy.

Figure 17.8 The Add Service Reference dialog with the correct service selected and namespace entered

Find the service (use the Discover button if it's in the same solution; otherwise, just enter the URL in the Address field), and then change the namespace to `Services`. The namespace will be used in the generated code, so it's important that it's something you can live with. If your app is hitting a number of different services, you can still use the same namespace as long as there are no naming collisions with the service class names themselves.

Once you have the service reference set up, the code to call it can be as simple as that shown here.

Listing 17.5 Calling the service from the MainPage.xaml.cs code-behind

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
  var client = new Services.MessageServiceClient();

  var response = await client.GetMessagesAsync();

  foreach (var message in response)
  {
    Debug.WriteLine(message.Text);
  }
}
```

Create proxy client

Call service method

Display results

Note that I said "can be." Calling the service from the code-behind is certainly an easy way to get started, but it's also not a best practice because it will quickly lead to a very messy and difficult-to-test-and-maintain solution. In fact, you won't even bother doing any UI binding here because I've found binding to things in the code-behind to not be worth the effort.

Instead, you're going to properly structure the code using a lightweight MVVM implementation—one of my favorite approaches.

17.4 Structuring your client code using MVVM

It's safe to say that most applications don't use the debug window as their primary form of output. Also, although many applications put all the code in the code-behind, as I've mentioned, that's not a great approach, especially when you start adding complexities like networking. For those reasons, you'll now modify the application to give it a little proper structure and to display its contents using the Windows app UI itself.

In previous chapters, you've used MVVM but through a toolkit. You can continue to use a toolkit here, but this chapter is also a good place to show how, in the context of networking, to do a little MVVM without using anything but the built-in types. That said, I don't want to fill up the chapter with a discussion of implementing `ICommand` (something most people don't do manually), so I'll follow a lighter-weight route using event handlers and function calls.

For structure, you'll add a very basic viewmodel that will expose the data, create the user interface XAML, and then use data binding to update the values on display.

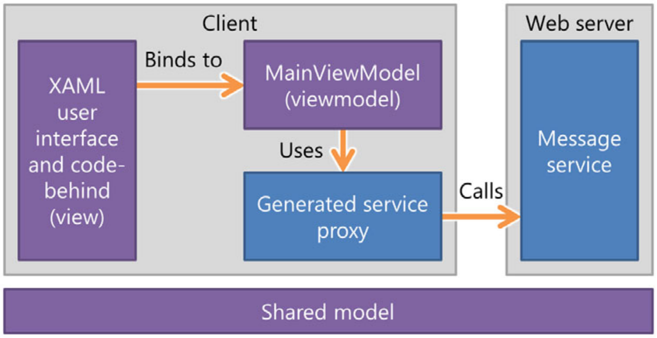


Figure 17.9 A basic MVVM implementation as used in this example. The model is shared between the client and server, the view is the client-side XAML and its code-behind, and the viewmodel is the class we just created on the client.

Following best practices, you'll also move the code that interfaces with the network to a separate class. Although this may at first glance seem like overkill for downloading the HTML file, you'll follow this structure for all the remaining service code, where it will really pay off. Figure 17.9 shows the structure you'll implement.

17.4.1 Creating the viewmodel

The first step is to create the viewmodel. This is a class that will contain every function the UI will use to interact with the rest of the system. It also will contain, quite importantly, all the data the UI will need to bind to, exposed in a binding-friendly form. More complex systems may not want directly called functions but may instead use commands for loading the data. We'll keep it simple here, while still gaining most of the benefits of the MVVM approach.

In the client project, add a folder named ViewModels. In that folder, create a class named `MainViewModel.cs`. This will be, as you probably suspected, the viewmodel you'll bind the UI to. The code for this class is shown here.

Listing 17.6 `MainViewModel` code

```
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using NetworkingClient.Model;

namespace NetworkingClient.ViewModels
{
  class MainViewModel : INotifyPropertyChanged
  {
    private ObservableCollection<InstantMessage> _messages =
            new ObservableCollection<InstantMessage>();

    public ObservableCollection<InstantMessage> Messages
    {
      get { return _messages; }
      set
      {
```

ViewModel class with INotifyPropertyChanged

Collection of messages

```
    if (_messages != value)
    {
     _messages = value;
     NotifyPropertyChanged("Messages");
    }
   }
  }

  public async void LoadMessages()
  {
   var client = new Services.MessageServiceClient();
   var response = await client.GetMessagesAsync();

   Messages = response;
  }

  public event PropertyChangedEventHandler PropertyChanged;
  protected void NotifyPropertyChanged(string propertyName)
  {
   if (PropertyChanged != null)
    PropertyChanged(this,
      new PropertyChangedEventArgs(propertyName));
  }
 }
}
```

Call to load the messages

INotifyPropertyChanged implementation

The viewmodel has two main parts: the collection of messages and the call to load the messages. If you've developed similar code using callbacks (for example, in Silverlight 5 without the async toolkit), you'll see that the `LoadMessages` call is quite a bit less complex, thanks to the `await` and `async` keywords.

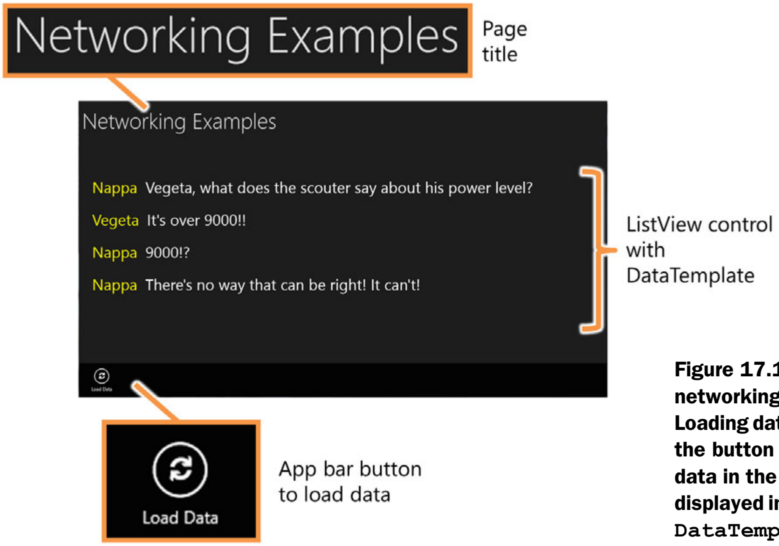
Refactoring service calls out of the viewmodel

An even better way to structure the client is to keep the viewmodel but have it call into a client-side service layer. This class, which may be a singleton, a static class, a class injected through Dependency Injection, or even a regular-old class, handles all the network calls. In the MVVM examples in previous chapters, this was a Service class.

Having the service proxy in a separate class abstracts it away from your viewmodel, making it much easier for you to change the services implementation, especially when testing and debugging.

Once my applications move beyond the most basic scenarios, I follow this approach.

Even if you don't plan on working with SOAP services in this chapter, you may want to build this viewmodel—you'll use it for the RESTful services as well. Similarly, the UI will be reused for both examples.


Networking Examples
Page title
Networking Examples
ListView control with DataTemplate
Load Data
App bar button to load data

Nappa Vegeta, what does the scouter say about his power level?
Vegeta It's over 9000!!
Nappa 9000!?
Nappa There's no way that can be right! It can't!

Figure 17.10 The finished networking example application. Loading data is accomplished via the button on the app bar. The data in the middle of the page is displayed in a `ListView` using a `DataTemplate`.

17.4.2 Creating and wiring up the user interface

The UI for this example will be extremely simple. You'll have a single app-bar button that will load all the messages into a list. The list will be a `ListView` control as used in most of the default templates. The function calls and event handlers will be written in the code-behind. Figure 17.10 shows the result you're aiming for.

The UI consists of just three parts:

- *The app title*—Displays the title for the application. We have no navigation this time around.
- *The `ListView`*—Shows the data from the service call.
- *The app bar*—Where all functionality goes.

The following listing contains the XAML to implement the UI shown in figure 17.9.

Listing 17.7 XAML UI for the MainPage.xaml view

```
<Page x:Class="NetworkingClient.MainPage"
  IsTabStop="false"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:NetworkingClient"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Page.BottomAppBar>
    <AppBar IsSticky="True" IsOpen="True">
      <Grid>
        <StackPanel Orientation="Horizontal">
```

AppBar with button

```
          <Button Style="{StaticResource RefreshAppBarButtonStyle}"
                  AutomationProperties.Name="Load Data"
                  Click="OnLoadDataClick"/>
        </StackPanel>
      </Grid>
    </AppBar>
  </Page.BottomAppBar>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <TextBlock x:Name="pageTitle" Text="Networking Examples"
               Margin="10,32,0,10" Grid.Row="0"
               Style="{StaticResource PageHeaderTextStyle}"/>
    <ListView x:Name="ResultsList" Margin="20, 100, 20, 20"
              Grid.Row="1"
              ItemsSource="{Binding Messages}">
      <ListView.ItemTemplate>
        <DataTemplate>
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding From}"
                       FontSize="40" Margin="10"
                       Foreground="Yellow" />
            <TextBlock Text="{Binding Text}"
                       FontSize="40" Margin="10"/>
          </StackPanel>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
  </Grid>
</Page>
```

App title

Messages from viewmodel

Bound DataTemplate

The XAML in this listing defines an `AppBar` at the bottom of the page. By default, the `AppBar` is both visible and sticky—you'll need to manually dismiss it with a swipe, keyboard shortcut, or a right-click. The style for the `AppBar` button is defined in the Common\StandardStyles.xaml resource dictionary. *If it is commented out, uncomment it now, or copy and paste it to app.xaml.* The markup also includes a standard page title and then a `ListView` for displaying the message data.

You won't be able to compile the project until you add the missing `OnLoadDataClick` event handler into the code-behind. You'll also need to wire up the `MainViewModel` to the `DataContext`, also in the code-behind. The next listing has everything needed for MainPage.xaml.cs.

Listing 17.8 Code-behind for MainPage.xaml.cs

```
using System;
using NetworkingClient.ViewModels;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
```

```
namespace NetworkingClient
{
  public sealed partial class MainPage : Page
  {
    private MainViewModel _vm = new MainViewModel();

    public MainPage()
    {
      this.InitializeComponent();
      DataContext = _vm;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    { }

    private void OnLoadDataClick(object sender, RoutedEventArgs e)
    {
      _vm.LoadMessages();
    }
  }
}
```

Create MainViewModel instance

Set Page DataContext

Forward button click to viewmodel

Because the majority of the functionality is encapsulated in the `MainViewModel` class, the code-behind is really lightweight. All it does (in addition to the usual page initialization) is create an instance of the viewmodel, set it as the `DataContext`, and forward the button click to the `MainViewModel`.

Run the application now, and click the app-bar button. You should see the data load into the `ListView` on the page. If you set a breakpoint in the `LoadMessages` method of the viewmodel, you can step through the code as it executes.

SOAP services aren't as common as they once were, but you have to admit that they are really simple to use. That ease of use comes not from SOAP itself but from the tooling built into Visual Studio and the Add Service Reference functionality.

One great thing about SOAP is as an RPC (Remote Procedure Call)-style API, all the operations follow the same pattern, whether you're inserting data, retrieving data, or doing something completely different. As you'll see in the next section, not all APIs work that way.

The application structure offered by the MVVM pattern really helps with cleaning up the binding and code-behind code. As you'll see in a moment, you won't have to change a thing in your UI to perform a service call using RESTful services, even though the networking code is quite different.

17.5 Consuming data from RESTful services

Representational State Transfer, or REST, means several things; in this case, it refers to the approach of making services accessible through a set of simple URIs and HTTP verbs. Before the days of web services and stateful web applications, most everything on the web was RESTful, meaning that all traffic over HTTP used one of the HTTP verbs to define its purpose, and calls were complete without requiring server-side state. Over the years, the use of these verbs dwindled down, with nearly all traffic using only the GET and POST verbs for requesting a page and submitting form data, respectively. Over the past few years there's been a trend toward moving from complex web services to a much simpler framework.

In the previous section we looked at SOAP services. SOAP is a nice way to handle remote procedure calls, but it suffers from being a very different protocol implemented on top of HTTP. Most caching and optimization infrastructures need to have code specifically optimized for SOAP, rather than being able to reuse what they're already doing for pages and other resources on the web.

Just as SOAP services were the thing in early 2000, RESTful services are becoming the popular way for exposing data on the internet and intranet. They are insanely simple to use and friendly to all types of consumers. You may recall with SOAP that if your platform of choice didn't have a good SOAP toolkit, you had to write a lot of code to create and parse SOAP envelopes. The overhead for RESTful services is so much lower that all you really need is a way to call URLs.

Many web service providers incorrectly use the term *REST* to mean any service that isn't SOAP. The main thing to realize is that the URI, and possibly the HTTP verb, may change depending on the action being performed. Typically, a creator of RESTful services will try to follow an intuitive structure where the URI first contains a type followed by an instance. For example, a URI with the structure http://www.arestfuldomain.com/Users might return an array of user records, whereas the URI http://www.arestfuldomain.com/Users/JohnSmith might return a single user record for John Smith.

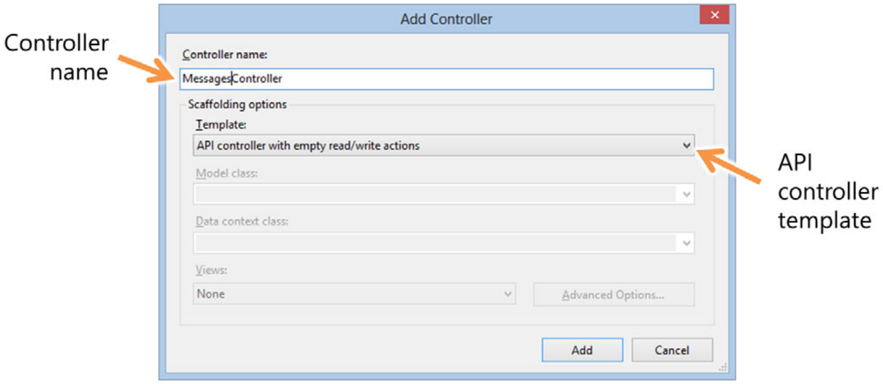
In this section you'll create a RESTful service using the ASP.NET Web API. The Web API (as it is commonly referred to) is a controller-based approach to building RESTful services. It builds on the foundation of MVC and the MVC patterns and conventions. The service will initially be used only for retrieving data—we'll get to updating and deleting data later. The last thing we'll look at in this section is how to control the type of data that's sent to you. Sometimes you may prefer XML; other times you may want JSON. With the client-side .NET objects and the WCF Web API on the server, you have complete control over which format you get.

17.5.1 Creating the RESTful service

If you skipped over the SOAP section, go ahead and load up the REST starter project from the supplied source code for this chapter. You'll use the UI and MVVM code as well as the MVC project from the SOAP example here.

Before you can write any client-side code, you need to add the RESTful service to the server-side project. The new MVC 4 templates make this extremely easy to do.

In the Site MVC project, right-click the Controllers folder and select the option to add a new controller. Name the controller `MessagesController`, and when prompted,



Controller name: MessagesController

Scaffolding options

Template: API controller with empty read/write actions

Model class:

Data context class:

Views: None

Figure 17.11 In the MVC project Controllers folder, create the new controller naming it `MessagesController` and use the "API controller with empty read/write actions" template.

change the template to the "API controller with empty read/write actions" template. Figure 17.11 shows the appropriate settings in the dialog.

The controller will be prepopulated with a string-based shell of a template. You won't use strings here; instead you want to use the `Message` class you created earlier. Ignoring the `Post`, `Put`, and `Delete` methods for the moment, change the two `Get` methods so the controller class looks like the following listing.

Listing 17.9 The `MessagesController` with `Get` methods in place

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using NetworkingClient.Model;

namespace Site.Controllers
{
  public class MessagesController : ApiController
  {
    private static List<InstantMessage> _data =
        new List<InstantMessage>()
      {
        new InstantMessage() { From="Pilot", To="CapAvatar",
            Text="A message from the Gamilon ship, sir:",
            Timestamp=DateTime.Parse("6/1/2012 2:57pm"), Id=1 },
        new InstantMessage() { From="Pilot", To="CapAvatar",
            Text="'Earth fleet, we advise you to surrender now'",
            Timestamp=DateTime.Parse("6/1/2012 2:58pm"), Id=2 },
        new InstantMessage() { From="Pilot", To="CapAvatar",
            Text="What shall I tell them?",
            Timestamp=DateTime.Parse("6/1/2012 2:59pm"), Id=3 },
        new InstantMessage() { From="CapAvatar", To="Pilot",
            Text="Tell them they're idiots!",
```

MessagesController

Hardcoded data

```
            Timestamp=DateTime.Parse("6/1/2012 3:00pm"), Id=4 },
      };

    public IEnumerable<InstantMessage> Get()
    {
      return _data;
    }

    public InstantMessage Get(int id)
    {
      return _data.Where(m => m.Id == id).FirstOrDefault();
    }

    public void Post(string value)
    { }

    public void Put(int id, string value)
    { }

    public void Delete(int id)
    { }
  }
}
```

Get all messages

Get single message

Post, Put, and Delete methods

Just as with the SOAP example, this service uses hardcoded data. But unlike the SOAP example, you store the data in a static member variable. Why? Because you'll need to access it from several different methods before the chapter is through.

I've removed the stock comments from the file, but the method names are very intuitive. The two `Get` methods are automatically mapped to the GET HTTP verb. Similarly, `Post`, `Put`, and `Delete` are mapped to the POST, PUT, and DELETE verbs.

Testing RESTful services is easy to do. Right-click the fan.html file you created at the beginning of this chapter, and select View in Browser.[5] When the browser comes up, change the URL to the right of the port number to be /api/messages. On my installation, the full URL looks like this: http://localhost:4621/api/messages.

If you're using Internet Explorer, you won't be able to view the resulting messages.json JSON (JavaScript Object Notation) file directly in the browser. You can, however, save it somewhere and open it in Notepad. You'll end up with a file that looks like this (without line breaks):

```
[{"Id":1,"From":"Pilot","To":"CapAvatar",
  "Timestamp":"2012-06-01T14:57:00",
  "Text":"A message from the Gamilon ship, sir:"},
 {"Id":2,"From":"Pilot","To":"CapAvatar",
  "Timestamp":"2012-06-01T14:58:00",
  "Text":"'Earth fleet, we advise you to surrender now'"},
 {"Id":3,"From":"Pilot","To":"CapAvatar",
  "Timestamp":"2012-06-01T14:59:00",
  "Text":"What shall I tell them?"},
```

---

[5] You could also change the Site project to be the startup project and simply run the solution. If you do that, remember to change the Windows Store app to the startup project when you've finished testing the RESTful service.

```
{"Id":4,"From":"CapAvatar","To":"Pilot",
 "Timestamp":"2012-06-01T15:00:00",
 "Text":"Tell them they're idiots!"}]
```

This is the data that was returned from the first `Get` operation in the controller code. JSON is the default return format. When you call the resource from the client, you'll be able to control that.

Now modify the URL to add a `Message` ID to the end. For example, append `/1` to the URL you used to get all the messages. On my machine, the full URL ends up as http://localhost:4621/api/messages/1. This calls the second API controller method, because the conventions involved automatically map the `/1` to the parameter `id` in the method. The resulting JSON will include a single `InstantMessage` instance, in this case, the one with `id=1`.

Now that you've verified that the service works, it's time to wire up the client code and test it out from the client app.

17.5.2 Getting data from the service using the viewmodel

Unlike the Visual Studio support for SOAP services, RESTful services currently don't have any tooling support for creating client-side proxies. The reason for this is simple: RESTful services aren't self-describing, unlike SOAP services, which use WSDL (Web Services Description Language).

When you own both sides of the conversation, however, it's easy to get around this. You can use the same model-sharing approaches as you did with SOAP and simply do the serialization using built-in classes.

Crack open the MainViewModel.cs file and find the `LoadMessages` function. Replace that code with the code shown here.

Listing 17.10 RESTful client implementation of `LoadMessages`

```
public async void LoadMessages()
{
  var client = new HttpClient();
  var uri = new Uri("http://localhost:4621/api/messages");        Set service URL

  var response = await client.GetAsync(uri);                      Call Get operation

  var data = await response.Content.ReadAsStringAsync();          Get response data

  Debug.WriteLine(data);
}
```

This listing shows an easy way to call the service in your MVC project. Make sure you resolve the appropriate namespaces: You'll need `System.Diagnostics`, `System`, and `System.Net`. Run the application and click the Load Data button, and you should see the same JSON output you saw earlier but now in the debug window.

You're not loading data into the `ListView` just yet, though. To do that, you need to deserialize the data into the appropriate `InstantMessage` instances. But to deserialize, you need to know exactly what type of data to expect, and so far, you've been relying on the service's implementation that returns JSON by default. It's better to exert some control, but to do that, you need to use a different client class.

17.5.3 Specifying the acceptable data type

The `HttpClient` class is useful for basic GET, POST, PUT, and DELETE actions, but it doesn't offer much control over the process. Specifically, there's no access to the collection of HTTP headers to be sent up with the call.

Rather than use the `HttpClient` class, you'll use the `HttpWebRequest` class via the `WebRequest.CreateHttp` factory method. This class returned from this method provides access to the various headers to be sent up with the request. One of those headers is Accept which the ASP.NET Web API uses to determine what type of data it should send back. The next listing shows the code.

Listing 17.11 Loading data specifying the results should be in XML format

```
public async void LoadMessages()
{
  var uri = new Uri("http://localhost:4621/api/messages");

  var request = WebRequest.CreateHttp(uri);                  Create request

  request.Method = "GET";                                    Specify verb and
  request.Accept = "text/xml";                               return data type

  var response = await request.GetResponseAsync();           Call service

  var stream = response.GetResponseStream();                 Get data
  StreamReader reader = new StreamReader(stream);            stream

  var data = await reader.ReadToEndAsync();                  Read data
                                                             string
  Debug.WriteLine(data);
}
```

This version accomplishes the same thing the previous listing did, but instead of JSON, it returns the data in XML format. This happens because of two things:

- You set the Accept header to specify that you'll only accept XML.
- The ASP.NET Web API respects the HTTP Accept header when it decides which format to send back the results.

Not all RESTful APIs properly respect the Accept header. In those cases, you'll need to work with just the data formats they return. Oh, and send them an email explaining how they're breaking the internet.

Once you have a data format you understand, you can deserialize the data into objects that can be used in binding. You already have those objects defined in your model, so let's get the data into them.

17.6 Deserializing JSON and XML data

Deserialization is just the process of taking the raw data and creating objects based on it. When you create a SOAP client proxy using Add Service Reference, the deserialization code is created for you automatically. When working with anything else, you need to handle the deserialization yourself.

You have several options, because the returned data is just a string of characters: You can manually process the data using string manipulation or helpers like LINQ to XML, or you can use built-in deserializers. Depending on the original structure and how well formed the data is, you may find you have to do the manual deserialization from time to time. Such is the nature of the web.

In this section, we'll look at the two automated forms of serialization: the `XmlSerializer` and the `DataContractJsonSerializer`. In both cases, you'll use the appropriate serializer to read the data from the ASP.NET Web API RESTful service you created earlier.

17.6.1 XML deserialization using XmlSerializer

Deserializing XML in .NET is a well-worn area. Generally, the only area that continues to trip people up is XML namespaces. Oftentimes, especially when dealing with automatically serialized data from a service, you don't know what namespace to specify. That's easily solved by inspecting the XML in the debug window or through a tool such as Fiddler.

Once you identify the necessary namespace(s), you can add them using the `XmlSerializer` constructor.

The next listing shows how to use the `XmlSerializer` to deserialize data from the Web API service. Notice that you're able to specify XML as the required format via the Accept header.

Listing 17.12 Deserializing with the `XmlSerializer`

```
public async void LoadMessages()
{
  var uri = new Uri("http://localhost:4621/api/messages");
  var request = WebRequest.CreateHttp(uri);

  request.Method = "GET";
  request.Accept = "text/xml";

  var response = await request.GetResponseAsync();
  var stream = response.GetResponseStream();

  var deserializer = new XmlSerializer(typeof(InstantMessage[]),
     "http://schemas.datacontract.org/2004/07/NetworkingClient.Model");

  var messages = (InstantMessage[])
     deserializer.Deserialize(stream);
```

Create HTTP request

Specify GET operation and XML format

Deserialize the data

```
  _messages.Clear();
  foreach (InstantMessage msg in messages)
    _messages.Add(msg);
}
```

Add objects to the collection

This listing shows how to specify an XML return format and then use the `XmlSerializer` to process the returned data. Note the full namespace passed into the `XmlSerializer` constructor—you can figure out the correct namespace simply by inspecting the returned data. Without this namespace declaration, the data won't deserialize.

Now run the application. It should behave just as it did in the SOAP example, except the results will look slightly different because the RESTful service data differs from the SOAP example.

Dynamic objects

Because you own both halves of the communication, it's easy to reuse the model classes between the server and the client. In Silverlight, it was necessary to have strongly typed classes to bind to. In WPF, and now also in WinRT XAML, that's not necessary.

Should your service return data that's more dynamic in nature, you don't need to create strongly typed classes on the client. Instead, WinRT XAML supports binding to dynamically created objects.

Dynamic, anonymous objects are generally created via LINQ statements. In those cases, you'll manually parse the data and create the anonymous object on the fly.

This is something to consider when you look at how to structure your model, because it could save you a fair bit of work, especially with highly dynamic APIs.

XML is a common data format, but as APIs move to better support JavaScript (or are created initially just for JavaScript developers), you'll find JSON becoming increasingly the dominant data format. Luckily, you can process that just as well as XML.

17.6.2 JSON deserialization

With the exception of date processing,[6] and the usual lack of namespace issues, JSON deserialization is almost identical to XML deserialization. In general, JSON is a lighter-weight format, so if you need to shave every last extra byte from the wire, JSON will likely result in smaller payloads. In typical use, this doesn't make much difference, but for really large data sets on constrained or expensive communications pipes like 3g/4g/…, it can be worth it.

---

[6] JSON dates—I hates it, my precious! Seriously, what kind of programming language doesn't have a standardized format for dates built in from the start?

The next listing shows how to request JSON data from the service and then deserialize it into the `InstantMessage` instances.

Listing 17.13 Deserializing JSON

```
public async void LoadMessages()
{
  var uri = new Uri("http://localhost:4621/api/messages");
  var request = WebRequest.CreateHttp(uri);

  request.Method = "GET";
  request.Accept = "application/json";

  var response = await request.GetResponseAsync();
  var stream = response.GetResponseStream();

  var settings = new DataContractJsonSerializerSettings();
  string dateFormat = "yyyy-MM-ddTHH:mm:ss";
  settings.DateTimeFormat =
      new DateTimeFormat(dateFormat);

  var deserializer = new DataContractJsonSerializer(
      typeof(InstantMessage[]), settings);

  var messages = (InstantMessage[])deserializer.ReadObject(stream);

  _messages.Clear();
  foreach (InstantMessage msg in messages)
    _messages.Add(msg);
}
```

GET JSON formatted data

Specify date format. JSON hates dates.

Deserialize

Add objects to collection

Run the application, and you should see exactly the same results you saw in the previous example. The wire data format itself isn't important to the UI.

> **NOTE** Many people prefer to use JSON.NET, especially because that's what the ASP.NET team uses. For one thing, it does a much cleaner job of dealing with dates. You are certainly free to use that library in Windows Store apps. See http://json.codeplex.com/.

When you work with APIs on the public web, you'll almost certainly run into both XML and JSON. It's good to know that both are easily supported in Modern C# apps.

So far, you've dealt only with retrieving data from a service. Anyone who has written an application knows that consumption isn't the only way to interact with services on the web—you also need to add, update, and sometimes delete data.

17.7 Updating data using PUT, POST, DELETE, and more

With ASP.NET Web API RESTful services, and with the web in general, the HTTP verb decides what you're going to do. If you want to add an item, you usually use the PUT verb. If you want to update an existing item, you use the POST verb, although POST sometimes plays the role of PUT as well. If you want to delete a resource, you'll use the DELETE verb. As you've already seen, if you want to download data, you use the GET verb. Recently, the PATCH verb was adopted on some platforms to support partial updates.

Most of us only ever use GET and POST, because the web has adapted to work with those and still function. As the web has moved from just pages (issuing a DELETE on a page would be … bad) to APIs, these verbs have seen increased use. They were there from the beginning, though. PUT and DELETE are not new by any stretch of the imagination.

In this section, you'll implement PUT, POST, and DELETE in the web service and use them from the client. You'll build on the previous code when implementing these features.

UPDATING THE MESSAGE SERVICE

The first thing to do is to add the new verb handlers to the `MessagesController` in the Site MVC project. The methods were autogenerated with strings as parameters. Please note that they now take `InstantMessage` instances rather than strings. The updated code, just for these methods, is shown here.

Listing 17.14 `Put`, `Post`, and `Delete` methods in the `MessagesController`

Update with Post

```
public void Post(InstantMessage value)
{
  var item =
    _data.Where(im => im.Id == value.Id).FirstOrDefault();

  if (item != null)
  {
    item.Text = value.Text;
    item.Timestamp = value.Timestamp;
    item.To = value.To;
    item.From = value.From;
  }
}
```

Add with Put

```
public void Put(int id, InstantMessage value)
{
  bool exists =
    (_data.Where(im => im.Id == value.Id).FirstOrDefault() != null);

  if (!exists)
  {
    value.Id = id;

    _data.Add(value);
  }
}
```

Remove with Delete

```
public void Delete(int id)
{
  var item =
    _data.Where(im => im.Id == id).FirstOrDefault();

  if (item != null)
    _data.Remove(item);
}
```

There are a few details in this listing that may have you scratching your head. The first is that the `Post` method, as implemented, can't handle updating the ID of an item. That's an implementation detail but a limitation that most people can live with. The code could have been written to match on something else, or the item could have contained the original and new values, much like ADO always did with change tracking in the `RecordSet` class.

The second is that the `Put` method takes an ID as a parameter when the ID is already part of the value. This is just the nature of the PUT pattern—the resource you `Put` doesn't need to have an ID internal to its structure. Think of a file. A file itself doesn't need to contain a filename—that's an external identifier.

The third thing that may jump out at you is the complete lack of any error reporting. If any of the operations fails to proceed due to a bad ID or something else, you don't report that back to the client. That's okay. I'm sure you've written code like that a million times and don't need it repeated here. Consider it my personal nod to your awesome skills and experience, and not me trying to get out of writing error code.

You're not quite finished with the web project, however. By default, IIS doesn't allow PUT and DELETE verbs. If you're running the full version of IIS, you can configure this through the GUI. On IIS Express, which comes with Visual Studio, you'll need to add the entry in the next listing to the Web.config in the Site project. Be sure to add it into the correct node—you'll almost certainly already have `system.webServer` and `validation` present in the file.

Listing 17.15 Web.config changes for PUT and DELETE

```
<system.webServer>
  <validation validateIntegratedModeConfiguration="false" />
  <modules runAllManagedModulesForAllRequests="true" />
  <handlers>
    <remove name="ExtensionlessUrlHandler-Integrated-4.0" />
    <add name="ExtensionlessUrlHandler-Integrated-4.0"
         path="*." verb="*"
         type="System.Web.Handlers.TransferRequestHandler"
         resourceType="Unspecified" requireAccess="Script"
         preCondition="integratedMode,runtimeVersionv4.5" />
  </handlers>
...
</system.webServer>
```

All paths, all verbs

Now that PUT and DELETE won't be filtered out by the web server, you're ready to call the service from the client.

CALLING THE NEW FUNCTIONS FROM THE CLIENT

With the functions all ready and waiting on the server, you need to call them. To have this make even a little sense, you need to update the UI with some buttons that will let you trigger the add, update, and delete code.

The following listing has the updated XAML for just the app bar on MainPage.xaml. Replace the existing `AppBar` entry with this markup.

Listing 17.16 Updated `AppBar` XAML for the new functions

```
<Page.BottomAppBar>
  <AppBar IsSticky="True"
          IsOpen="True">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
      </Grid.ColumnDefinitions>
      <StackPanel Orientation="Horizontal">
        <Button Style="{StaticResource RefreshAppBarButtonStyle}"
                AutomationProperties.Name="Refresh Data"
                Click="OnLoadDataClick"/>
        <Button Style="{StaticResource AddAppBarButtonStyle}"
                AutomationProperties.Name="Add Item"
                Click="OnAddItemClick"/>
      </StackPanel>
      <StackPanel Grid.Column="1" HorizontalAlignment="Right"
                  Orientation="Horizontal">
        <Button Style="{StaticResource EditAppBarButtonStyle}"
                AutomationProperties.Name="Update Selected Item"
                Click="OnUpdateItemClick"/>
        <Button Style="{StaticResource DeleteAppBarButtonStyle}"
                AutomationProperties.Name="Delete Selected Item"
                Click="OnDeleteItemClick"/>
      </StackPanel>
    </Grid>
  </AppBar>
</Page.BottomAppBar>
```

Add item

Update item

Delete item

This will result in an app bar with buttons, as shown in figure 17.12. The commands that are independent of any selected item may be found to the left. The commands that are item-specific are on the right.

The next step is to update the MainViewModel with the appropriate code. The code in the viewmodel will serialize the data, build the URLs, and then call the service. The next listing has the code to handle the selected message, as well as the function that's used to send the requests to the server.

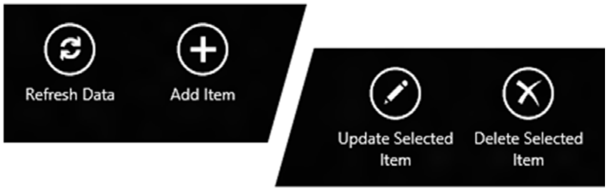


Figure 17.12 The app bar with the new Add, Update, and Delete buttons. I also changed the Load Data button to read Refresh Data. The item-specific commands are on the right; the globals are on the left. This is the opposite of the recommended approach as covered in chapter 11—an oversight I caught only as I was sending this chapter to the printer.

Listing 17.17 `SelectedMessage` and `SendRequest` viewmodel code

```
private InstantMessage _selectedMessage;
public InstantMessage SelectedMessage
{
    get { return _selectedMessage; }
    set { _selectedMessage = value;
```

Selected message

```
          NotifyPropertyChanged("SelectedMessage"); }
}

private const string RootMessagesServiceUrl =
    "http://localhost:4621/api/messages/";                      Root URL

private async void SendRequest(string verb,
                               InstantMessage message, string url)
{
  var uri = new Uri(url);
  var request = WebRequest.CreateHttp(uri);

  request.Method = verb;
  request.ContentType = "application/json";                   JSON format

  var requestStream = await request.GetRequestStreamAsync();

  var settings = new DataContractJsonSerializerSettings();
  string dateFormat = "yyyy-MM-ddTHH:mm:ss";
  settings.DateTimeFormat = new DateTimeFormat(dateFormat);

  var serializer = new DataContractJsonSerializer(
         typeof(InstantMessage), settings);                  Data serialization
  serializer.WriteObject(requestStream, message);

  var response = await request.GetResponseAsync();          Actual network call
}
```

This listing includes the `SelectedMessage` property, which is bound to from the UI (more on that in a moment) as well as the `SendRequest` method. Both the add and update functions have almost identical code for serialization and networking, so that code was factored out into the `SendRequest` method.

To support the `SelectedMessage` property in the viewmodel, there's one additional item to add to the XAML. In the `ResultsList` `ListView` in MainPage.xaml, add the `SelectedItem` and `SelectionMode` attributes:

```
<ListView x:Name="ResultsList"
          SelectedItem="{Binding SelectedMessage, Mode=TwoWay}"
          SelectionMode="Single"
          ...
```

This binds the `ListView`'s `SelectedItem` property to the `SelectedMessage` on the viewmodel. Because it's two-way binding, any time either the viewmodel or the `ListView` changes, the other will be updated. This is a very common MVVM approach.

The following listing has the actual add, update, and delete methods that go in the viewmodel class. Note that update and add use hardcoded changes, because we won't have a UI for entering data.

Listing 17.18 Add, update, and delete methods in the MainViewModel

```
public void UpdateSelectedMessage()                  Update
{
  if (SelectedMessage != null)
  {
    SelectedMessage.Text += " upd";
    SendRequest("PUT", SelectedMessage,
                 RootMessagesServiceUrl + SelectedMessage.Id);
  }
}
```

Add

```
public async void AddMessage()
{
  var message = new InstantMessage();

  message.Id = (int)(DateTime.Now.Ticks & 0xFFFF);
  message.Timestamp = DateTime.Now;
  message.From = await UserInformation.GetDisplayNameAsync();
  message.To = "Pete";
  message.Text = "Some random text from viewmodel";

  SendRequest("POST", message, RootMessagesServiceUrl);
}
```

Delete

```
public async void DeleteSelectedMessage()
{
  if (SelectedMessage != null)
  {
    var uri = new Uri(RootMessagesServiceUrl + SelectedMessage.Id);
    var request = WebRequest.CreateHttp(uri);

    request.Method = "DELETE";

    var response = await request.GetResponseAsync();
  }
}
```

This listing includes the update, add, and delete code. Of them, both `UpdateSelectedMessage` and `AddMessage` make use of the `SendRequest` method from the previous listing, because they both have to serialize data to be sent up to the server.

I threw in something extra there. In the `AddMessage` method, you may have noticed that I've wired up code to get the display name of the current user and use that as the "From" user.

Now, the workflow you're using here is only illustrative. Some applications will benefit from adding to the local collection and then batching changes to be sent to the server. The approach is entirely up to you. In this sample, the workflow is this: Add, update, or delete an item, and then refresh the data to view the changes.

To see that workflow in action, wrap up the changes with the code in the next listing. This code goes in MainPage.xaml.cs.

Listing 17.19 Wiring it together in the code-behind

Add item

```
private void OnAddItemClick(object sender, RoutedEventArgs e)
{
  _vm.AddMessage();
}

private async void ShowSelectSomethingMessage()
{
  var dlg = new MessageDialog(
    "Please select a message before selecting this option.",
    "Messages");
  await dlg.ShowAsync();
}

private void OnUpdateItemClick(object sender, RoutedEventArgs e)
{
  if (_vm.SelectedMessage != null)
    _vm.UpdateSelectedMessage();                             Update
  else                                                       selected item
    ShowSelectSomethingMessage();
}

private async void OnDeleteItemClick(object sender, RoutedEventArgs e)
{
  if (_vm.SelectedMessage != null)
  {
    var dlg = new MessageDialog(
      "As you sure you want to delete the selected message from '" +
          _vm.SelectedMessage.From + "'?",
      "Delete");

    var yesCommand = new UICommand("Yes");
    var noCommand = new UICommand("No");
    dlg.Commands.Add(yesCommand);
    dlg.Commands.Add(noCommand);

    var cmd = await dlg.ShowAsync();                          Confirm deletion

    if (cmd == yesCommand)
      _vm.DeleteSelectedMessage();                       Delete
  }                                                      selected item
}
```

The code in this listing is the glue between the UI XAML and the viewmodel. Its responsibility is user interaction such as confirming and forwarding button click events to the viewmodel.

At this point, you can run the application and add, update, and delete. Keep in mind that the workflow requires manually refreshing the data after each operation. If you run out of data, you can stop IIS in the taskbar and rerun the application—the data will return to its original hardcoded state.

By having this code in the viewmodel and deserializing the data into strongly typed .NET objects, you've kept the implementation code away from the UI. But in an application with multiple viewmodels, this can quickly get messy.

Refactoring networking code into a service proxy class

That URL building and serialization code in the viewmodel is annoying at best and brittle at worst. I don't like how it brings the service dependency so far down into the application. A better solution is to pull out the service-dependent code and put it into a separate class and call methods in that class from the viewmodel. I mentioned this in the discussion about SOAP, but the benefits in that example were marginal. Here, it's far more concrete.

(continued)

To do this, first, go ahead and remove the service reference you had for the SOAP service—you won't need it anymore, and I don't want there to be any confusion as to what code you're calling. Simply right-click the Services entry under the Service References folder, and select Delete.

Next, create appropriate functions in the proxy. You'll likely want to have awaitable `GetAllMessages`, `GetSingleMessage`, `UpdateMessage`, `AddMessage`, and `DeleteMessage` methods. Keep the root service URL in a single, easily set location, perhaps as a constant, perhaps as a constructor parameter.

Once you have the proxy class set up, change the viewmodel methods to simply call into it. This will simplify the viewmodel code and facilitate better reuse of the networking code in multipage applications where multiple viewmodels need access to the same services.

Working with RESTful services can be really exciting and rewarding. REST is turning out to be the de facto API style on the internet, used by the major social networking sites and many others. Its simple format, without the overhead and complexity of SOAP, makes it very appealing both for clients with built-in support for serialization like Modern Style C#, as well as those that are built from scratch.

Quite frankly, with its reliance on the standard HTTP verbs like GET, POST, PUT, and DELETE, it just works the way the web is supposed to. It's so easy to understand and so basic in implementation, it's even popular on tiny .NET Micro Framework boards like the Netduino and .NET Gadgeteer boards I use often when working with robotics, sensors, and side projects.

17.8 Summary

This chapter introduced you to the basics of networking, as well as to creating and consuming services. Regardless of the type of application you develop, you'll probably need to access a resource on the web or call some sort of service. You may create your own website, as we did at the start of this chapter. You may even use it just to download files. It's also quite possible, and quite likely, that you'll use existing public services as either a part of or the entirety of your connectivity requirements. Whatever you need to do, the code will remain the same.

SOAP services are the old standby. We've been using them for well over a decade now. In fact, SOAP service support was one of the early driving factors behind .NET. You'll still find SOAP services, both new and old, especially in the enterprise. Visual Studio tooling through Add Service Reference makes them incredibly easy to use.

RESTful services are quickly becoming the new hotness for service development. They're easy to consume from a variety of platforms, including manually in a browser or through a tool like Fiddler. Sure, there's no client-side proxy created for you like the tooling does for SOAP, but that's not a huge loss when you weigh it against the increased reach and lower overhead.

When it comes to consuming data from a REST service or storing data locally, the two most popular formats are XML and JSON. Windows Store apps can use the `XmlSerializer` or manual methods when parsing XML. I tend to use LINQ to XML when doing small tasks and the `XmlSerializer` when I have a serious set of objects to work with or when sharing model objects between the client and server.

JSON is an even simpler format than XML. Although you could also use manual methods for processing JSON, you can't beat the `DataContractJsonSerializer` for its ability to parse deep JSON objects quickly and with minimal code.

In this chapter, we also looked at a couple of my favorite techniques for application development: cross-compiling code between two platforms and organizing your code using the MVVM pattern. Cross-compilation makes it easy for you to share your model, in this case, between ASP.NET and your Windows Store client app. MVVM is a no-brainer, even in the simple form I showed in this chapter. If you plan to use binding (which you will), having your binding sources all wrapped up in a page-specific viewmodel makes it so much easier to work with and understand your code.

In the next chapter, the discussion of networking continues with a collection of some of the other interesting networking capabilities offered by .NET 4.5 and the Windows Runtime.

18

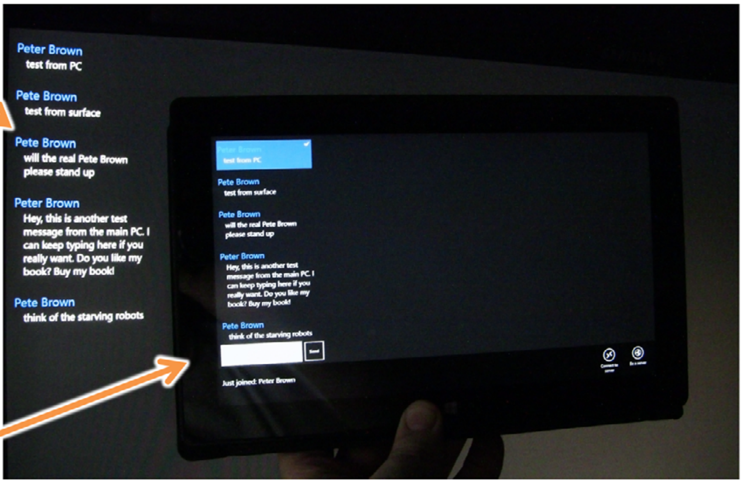
A chat app using sockets

This chapter covers

- Creating a socket server
- Connecting to a socket server
- Using TCP and UDP sockets

Higher-level networking approaches such as SOAP and RESTful services are great when you don't need to count milliseconds or when communication is primarily one way. But what about those times when you need to perform near real-time control of, say, a robot? How about synchronizing character or object movement for a game? Those are all performance-critical, often bidirectional, and sometimes peer-to-peer communications scenarios.

When apps need to communicate across a network as quickly and efficiently as possible, they use sockets. Socket communication is two-way communication between, in most cases, two endpoints. (Multicast/broadcast socket is the one-to-many or even many-to-many approach used in some other scenarios. I won't cover those approaches here because they aren't as commonly used, but I do build on the normal socket communication in this chapter.)



Desktop PC
"Peter Brown"

Peter Brown
test from PC

Pete Brown
test from surface

Pete Brown
will the real Pete Brown please stand up

Peter Brown
Hey, this is another test message from the main PC. I can keep typing here if you really want. Do you like my book? Buy my book!

Pete Brown
think of the starving robots

Just joined: Peter Brown

Surface in
my hand
"Pete Brown"

Figure 18.1 The app for this chapter running on my main PC as well as my Surface. The PC version has the same app bar as the Surface, but it's out of frame in the photo. The two machines are communicating over sockets. You'll have to trust me that this very large blob of dark print is, in fact, two separate screens running the app.

Sockets are what power the internet. Protocols such as HTTP are built on top of socket communication. Whenever you specify a port, you're specifying an endpoint for a socket, with the additional protocol built on top of it. If you want to implement your own protocol, you'll almost certainly start with socket communication.

In this chapter, you'll start building a socket-based communications app. The functionality introduced here will be simple peer-to-peer chat between two machines. Subsequent chapters will add even more game-like features that will build on the same communications and messaging infrastructure, but starting with chat makes it all easier to visualize and understand.

Figure 18.1 shows the peer-to-peer chat functionality of the app you'll build in this chapter. The app will work on all flavors of Windows 8, including Windows RT and the Microsoft Surface.

The app is both the client and the server, something that many don't realize can be done inside a Windows Store app. To make this work, you'll need to use two machines as the endpoints of the communication.

First, we'll turn to the MVVM pattern to help you structure the app. The functionality will be completely encapsulated within the viewmodel and will start with TCP streaming sockets. The UI that binds to the viewmodel will be kept simple in order to focus on learning the mechanics of socket communication. Once you get the basic chat app working, you'll refactor the code and add UDP socket support into the mix. By the end of this chapter, you'll have a simple chat app that can be used across two machines on a network. In my case, my two machines are my Windows 8 desktop PC and my Windows RT Surface.

Why two machines?

Two machines for a demo? This may seem like an attempt on my part to sell more licenses on behalf of my employer, but trust me, it's not. Really!

I was tempted to do my usual approach of using a .NET Micro Framework (NETMF) device (Netduino or Gadgeteer) but figured that would overflow the geek-o-meter for this book. You can definitely learn from the code even with a single Windows 8 machine, but sockets are point-to-point and you need two points for a conversation.

Why didn't I set up a console app on the desktop or have communication between two Windows Store apps on the same machine? You can make network calls to the loopback (127.0.0.1) if enabled in Visual Studio in the properties page for the project, but for actual deployed Windows Store apps, this is forbidden. To be very clear: Windows Store apps aren't allowed to open network connections to the same machine, even if it happens to work in Visual Studio. For that reason, I won't include examples for that scenario here.

If you'd rather go the NETMF route to, you know, control robots and blow things up, evict neighbors, and the like, I have some source code posted on my personal site as well as on the official .NET team blog.

18.1 Chat app viewmodel

When working with socket communication, a meaningful app can get complex quite quickly. So rather than tackle the entire app all at once, we'll start with the chat portion and refactor after that before adding more functionality in the next chapters.

A chat app is good for learning peer-to-peer connectivity. The message structure is simple, and the UI interaction patterns have already been explored elsewhere in this book. It'll look a little sparse at first—a good portion of the UI will be blank other than the chat column over on the left, as shown in the first figure in this chapter.

As I mentioned, this app will follow the MVVM pattern introduced earlier in this book. We'll again use Laurent Bugnion's MVVM Light toolkit in this app, because it's a huge time-saver when you want to use commanding and strongly typed property change notifications.

Figure 18.2 shows how the viewmodel fits into the picture. For this round, all the functionality is inside the viewmodel itself, because you want to focus on learning socket communication.

NOTE For those of you more experienced with MVVM, sitting on the edge of your seats screaming at this book because the sockets code is implemented directly in the viewmodel, I hear you. You know I wouldn't do that to you and just leave it out there. By the end of this chapter, you'll have all that sockets code factored out into its own service class. In the meantime, I recommend getting one of those little desktop Zen gardens.

All of the UI interaction for the chat functionality uses binding to communicate with the viewmodel. In fact, because of the naming convention for the viewmodel (it

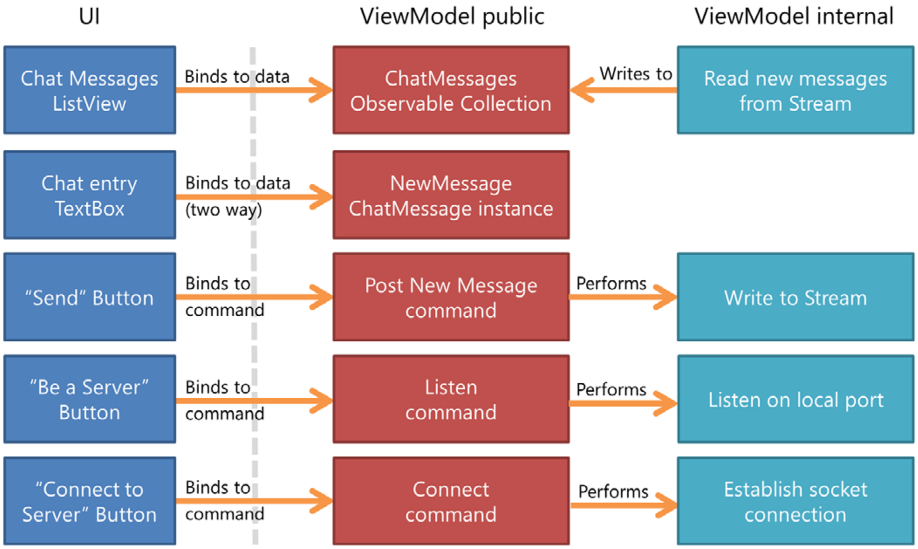

UI
ViewModel public
ViewModel internal
Chat Messages ListView
Binds to data
ChatMessages Observable Collection
Writes to
Read new messages from Stream
Chat entry TextBox
Binds to data (two way)
NewMessage ChatMessage instance
"Send" Button
Binds to command
Post New Message command
Performs
Write to Stream
"Be a Server" Button
Binds to command
Listen command
Performs
Listen on local port
"Connect to Server" Button
Binds to command
Connect command
Performs
Establish socket connection

Figure 18.2 The UI is bound to the `ViewModel` using commands for buttons and also using one-way and two-way data binding for input and the messages list. Currently, all of the socket communication is also inside the `ViewModel`, but that will change before the end of this chapter.

matches the page name "MainPage" with "MainViewModel"), you're able to use the MVVM Light viewmodel locator implementation to automatically wire up the UI. The result is code-behind with no lines of code other than those from the stock template. An empty code-behind file is generally not the ultimate or most important goal, but when it comes about as the result of good app structure, you'll find you can better design the UI and better test the app functionality.

To start, create a new project named SocketApp, using the MVVM Light XAML/C# template for Windows Store apps. If you're unfamiliar with this template, please refer to chapter 9 on MVVM and controls.

The main tasks we'll look at in this section are building out the skeleton of the `MainViewModel` class and creating the `ChatMessage` model. The `MainViewModel` will have several placeholder methods, which you'll complete later in this chapter.

18.1.1 The MainViewModel class

The `MainViewModel` class is what the UI uses to communicate with the rest of the app. It's where the command instances are located and where the bindable properties are surfaced to the UI. Open up the `MainViewModel` class source file in the ViewModel folder and replace its contents with what's shown in this listing.

Listing 18.1 The skeleton `MainViewModel`

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using GalaSoft.MvvmLight.Threading;
using SocketApp.Model;
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;

namespace SocketApp.ViewModel
{
  public class MainViewModel : ViewModelBase
  {
    private const string PortOrService = "5150";

    private StreamSocket _socket;
    private StreamSocketListener _listener;

    public MainViewModel()
    {
      ChatMessages = new ObservableCollection<ChatMessage>();
      ConnectionStatus = "Not connected.";
      ServerAddress = "pete-surface64";

      CreateNewMessage();

      PostNewMessageCommand = new RelayCommand(
        () => PostNewMessage(), () => CanPostNewMessage());

      ListenCommand = new RelayCommand(
        () => Listen(), () => CanListen());

      ConnectCommand = new RelayCommand(
        () => Connect(),() => CanConnect());
    }

    public ObservableCollection<ChatMessage> ChatMessages { get; set; }

    private ChatMessage _newMessage;
    public ChatMessage NewMessage
    {
      get { return _newMessage; }
      set { Set<ChatMessage>(() => NewMessage, ref _newMessage, value); }
    }

    public RelayCommand PostNewMessageCommand { get; private set; }

    private void CreateNewMessage()
    {
      if (NewMessage != null)
        NewMessage.PropertyChanged -= NewMessage_PropertyChanged;
```

Socket server port number

Input/output socket

Socket listener

Server name (change this)

Chat messages

Message entry

Create empty message

```
      NewMessage = new ChatMessage();
      NewMessage.PropertyChanged += NewMessage_PropertyChanged;
    }

    void NewMessage_PropertyChanged(object sender,
         PropertyChangedEventArgs e)
    {
      if (e.PropertyName == "Message")
        PostNewMessageCommand.RaiseCanExecuteChanged();
    }

    public async void PostNewMessage() { }
    public bool CanPostNewMessage() { return true; }

    private string _serverAddress;
    public string ServerAddress
    {
      get { return _serverAddress; }
      set { Set<string>(() => ServerAddress, ref _serverAddress, value); }
    }

    private string _connectionStatus;
    public string ConnectionStatus
    {
      get { return _connectionStatus; }
      set { Set<string>(() => ConnectionStatus,
            ref _connectionStatus, value); }
    }

    public RelayCommand ConnectCommand { get; private set; }
    public async void Connect() { }
    public bool CanConnect() { return true; }

    public RelayCommand ListenCommand { get; private set; }
    public async void Listen() { }
    public bool CanListen() { return true; }
  }
}
```

Post chat message

Connect to server

Be a server

This single viewmodel has all the endpoints required for wiring up the UI. Several of the methods are placeholders (such as the code to be a server or connect to one) and will be implemented throughout this section.

The viewmodel has a hardcoded server name. In my case, the server is my Microsoft Surface. Replace that with the machine name or IP address of the machine you intend to connect to. The port number is also up to you, as long as you pick something that's out of the restricted range of well-known ports. If you have firewall issues on your network, you'll find that changing it to port 80 will work, as long as the server machine isn't running a web server of any sort.

The message list and message entry functionality works just as you've seen in the previous chat example in this book (chapter 9 on MVVM and controls), so I won't go into detail on that pattern here.

18.1.2 *ChatMessage model class*

The viewmodel includes a couple references to the `ChatMessage` class, using the following listing as the body of a class named `ChatMessage` in the Model folder.

Listing 18.2 The `ChatMessage` model class

```
using GalaSoft.MvvmLight;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SocketApp.Model
{
  public class ChatMessage : ObservableObject
  {
    private string _message;
    public string Message
    {
      get { return _message; }
      set { Set<string>(() => Message, ref _message, value); }
    }
  }
}
```

Message text

The `ChatMessage` class contains a single property in this initial version: the text of the message. Using only this single property will help you keep the sockets messaging format simple and understandable for this first version.

The viewmodel and model are two of the most importance pieces for this app. The viewmodel, in particular, is where all the sockets action will happen. Now that you have everything that the UI binds to, it's time to create the UI itself.

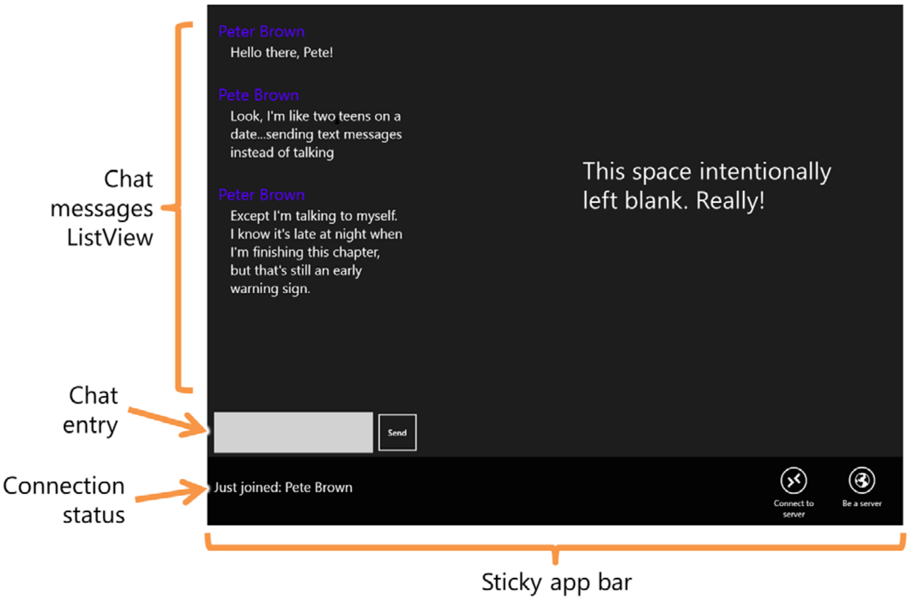
18.2 *The user interface*

The app you create in this chapter will be the chat portion of a larger peer-to-peer app. Because you want as much space as possible for app content, there will be no title portion. In addition, because there's only a single page in this app, there's no need for a navigation button at the top left.

The UI will make use of data binding to the viewmodel both for the list of chat messages as well as for the entry of the new chat message. It will also use data binding to update the connection status in the app bar and command binding for the three buttons.

Figure 18.3 shows what the UI will look like once you've completed the app.

In this section, you'll build out the XAML user interface for this app. You'll start with a simple skeleton and then add in the styles and resources used for the buttons and text. From there, you'll create an app bar with two buttons and a couple of `TextBlock` elements. The app bar will be sticky and visible, so you don't have to manually



Peter Brown
Hello there, Pete!

Pete Brown
Look, I'm like two teens on a date...sending text messages instead of talking

Peter Brown
Except I'm talking to myself. I know it's late at night when I'm finishing this chapter, but that's still an early warning sign.

This space intentionally left blank. Really!

Just joined: Pete Brown

Figure 18.3 A cropped view of the UI, showing the main elements plus the app bar

show it. Next, you'll create the `ListView` for the chat messages and the controls that allow for message entry. We'll wrap up this section with a very sparse set of visual states for the different page view states.

18.2.1 *XAML skeleton*

This app has only a single page: MainPage.xaml. You're not going to do anything specific to support portrait and snapped views in this version, but snapped view will "just work." Crack open MainPage.xaml and replace its contents with what you see in the next listing. This will serve as the starting structure for the interface.

Listing 18.3 MainPage.xaml skeleton

```
<common:LayoutAwarePage x:Class="SocketApp.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:common="using:SocketApp.Common"
  xmlns:ignore="http://www.ignore.com"
  mc:Ignorable="d ignore"
  d:DesignHeight="768"
  d:DesignWidth="1366"
  DataContext="{Binding Main, Source={StaticResource Locator}}">

  <!-- Styles and resources go here -->

  <!-- App Bar goes here -->
```

ViewModelLocator

Styles and resources

App bar

```
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    <!-- Content goes here -->                    Content

    <!-- Visual States Go Here -->                    Visual states
  </Grid>
</common:LayoutAwarePage>
```

For this chapter, I make use of the `ViewModelLocator` provided in MVVM Light. You can see its reference as the data context for this page.

The comments in this listing are placeholders for content you'll add in the next several listings.

18.2.2 Styles and resources

The page includes styles for the buttons as well as colors that will be used for text and other elements. These are all included in the resources section of the page.

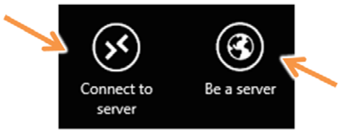
The first of those is the styles and resources local to this page. The following listing contains the XAML to place at that spot.

Listing 18.4 MainPage.xaml styles and resources

```
<Page.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="Skins/MainSkin.xaml" />
    </ResourceDictionary.MergedDictionaries>

    <SolidColorBrush x:Key="AccentBrush" Color="#FF220088" />
    <SolidColorBrush x:Key="HighlightBrush" Color="#FF5500FF" />

    <Style x:Key="WorldAppBarButtonStyle" TargetType="ButtonBase"
           BasedOn="{StaticResource AppBarButtonStyle}">
      <Setter Property="AutomationProperties.AutomationId"          Server app
              Value="WorldAppBarButton" />                          bar button
      <Setter Property="AutomationProperties.Name"
              Value="World" />
      <Setter Property="Content"
              Value="&#xE128;" />                                   Connect app
    </Style>                                                        bar button

    <Style x:Key="RemoteAppBarButtonStyle" TargetType="ButtonBase"
           BasedOn="{StaticResource AppBarButtonStyle}">
      <Setter Property="AutomationProperties.AutomationId"
              Value="RemoteAppBarButton" />
      <Setter Property="AutomationProperties.Name" Value="Remote" />
      <Setter Property="Content" Value="&#xE148;" />
    </Style>
  </ResourceDictionary>
</Page.Resources>
```

The styles used in the resources section of the page are for the buttons. The two app bar button styles are copied directly from the commented-out, template-provided styles in app.xaml. The colors will be used later in this chapter and in the next.

Click or tap to connect to existing server.



Click or tap to make this machine the server.

Figure 18.4 The app bar buttons for the chat app

18.2.3 App bar buttons

This app contains only two app bar buttons, shown in figure 18.4. The first is used when you want to connect to the app running as a server on another machine. The second is used when you want the machine itself to be the server. On any given machine, you'll choose only one of these options, not both. But you don't do any validation of that or enable/disable the buttons (via the commands). That's something you could easily add in if you'd like—it would be handled 100% in the viewmodel.

The app bar also includes some `TextBlock` elements on the left, one of which you'll use to display the connection status.

The app bar buttons, the status text, and the app bar that contains them are all shown in the following listing. Place this markup in the page section reserved for the app bar.

Listing 18.5 MainPage.xaml app bar

```
<Page.BottomAppBar>
  <AppBar IsSticky="True" IsOpen="True">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>

      <StackPanel Orientation="Vertical" Grid.Column="0"
                  HorizontalAlignment="Left"
                  VerticalAlignment="Center">
        <TextBlock x:Name="ConnectionStatus" FontSize="20"
                   TextWrapping="Wrap"
                   Text="{Binding ConnectionStatus}" />
        <TextBlock x:Name="IPAddressDisplay" />
      </StackPanel>

      <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Right"
                  Grid.Column="1">
        <Button x:Name="ConnectToServer"
                Command="{Binding ConnectCommand}"
                Style="{StaticResource RemoteAppBarButtonStyle}"
                AutomationProperties.Name="Connect to server" />
        <Button x:Name="BeAServer"
                Command="{Binding ListenCommand}"
                Style="{StaticResource WorldAppBarButtonStyle}"
                AutomationProperties.Name="Be a server"/>
```

Connection status

Buttons on right

Connect app bar button

Server app bar button

```
      </StackPanel>
    </Grid>
  </AppBar>
</Page.BottomAppBar>
```

Each of the buttons uses command binding to communicate with the viewmodel. Note also that the styles used are the ones you added to the resources section earlier.

18.2.4 Chat app content

Finally, we get to the action part of the app: the chat messages list and entry UI. For the chat app, the content consists of the `ListView` containing the chat messages, a `TextBox` to type the message, and a button to send the message. The markup to place in the "content" placeholder on the page is shown here.

Listing 18.6 MainPage.xaml content

```
<Grid Margin="0,0,0,110">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="320" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>

  <Grid Grid.Column="0"
        Margin="10,10,10,0">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ListView Grid.Row="0" Margin="0,0,0,10"
              ItemsSource="{Binding ChatMessages}">
      <ListView.ItemTemplate>
        <DataTemplate>
          <Grid Margin="0,5,0,5">
            <TextBlock FontSize="20" TextWrapping="Wrap"
                       Text="{Binding Message}" />
          </Grid>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>

    <Grid Grid.Row="1">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>

      <TextBox x:Name="ChatEntry" MaxLength="512"
               Text="{Binding NewMessage.Message, Mode=TwoWay}"
               TextWrapping="Wrap" FontSize="14"
               Grid.Column="0" Height="60"
               HorizontalAlignment="Stretch" />
```

List of chat messages

Chat entry

Send message

```
      <Button Content="Send" Command="{Binding PostNewMessageCommand}"
              Margin="5,0,0,0" FontSize="12"
              VerticalAlignment="Stretch"
              Grid.Column="1" />

    </Grid>
  </Grid>
</Grid>
```

This listing shows how binding is used for the button commands. One interesting note here: The `MainViewModel` code for the `CanPostNewMessage` function always returns true. This is because there's currently no easy way to update the command for each letter typed in the `TextBox`. In other XAML-based UI, you'd set the `UpdateSourceTrigger` to `PropertyChanged`, but that's not yet available in WinRT XAML. You may have seen this in chapter 9 where you had to tab off the chat field in order to enable the button (or click the button twice).

The final bit of markup is for the visual states. As I mentioned earlier, you're not doing anything special to handle the different orientations and states, but the UI is simple enough that it works as is. The following listing has the XAML to place in the visual states section.

Listing 18.7 MainPage.xaml Visual States

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ApplicationViewStates">
    <VisualState x:Name="FullScreenLandscape" />
    <VisualState x:Name="Filled" />
    <VisualState x:Name="FullScreenPortrait" />
    <VisualState x:Name="Snapped" />
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

View states

The UI is in place, and most of the rest of the structure of the app is there. But the `MainViewModel` is full of placeholders for things like connecting to a server and listening for new connections.

Note that one thing you didn't have to do is open up the code-behind. Everything in this app (for at least the chat functionality) will be handled via binding to viewmodel data items and commands.

Nothing in the UI should be surprising to you, because it's all stuff we've covered in previous chapters, simply reapplied here. Binding of data and commands is an especially important concept to learn. One of the commands will be to enable the app as a socket server, so let's cover that next.

18.3 Listening for connections

Sockets require that at least one endpoint be set up to listen for new connections ahead of time. In a client/server-based solution, like the browser and a web server, the server is what listens. Listening simply establishes the initial communication; after a

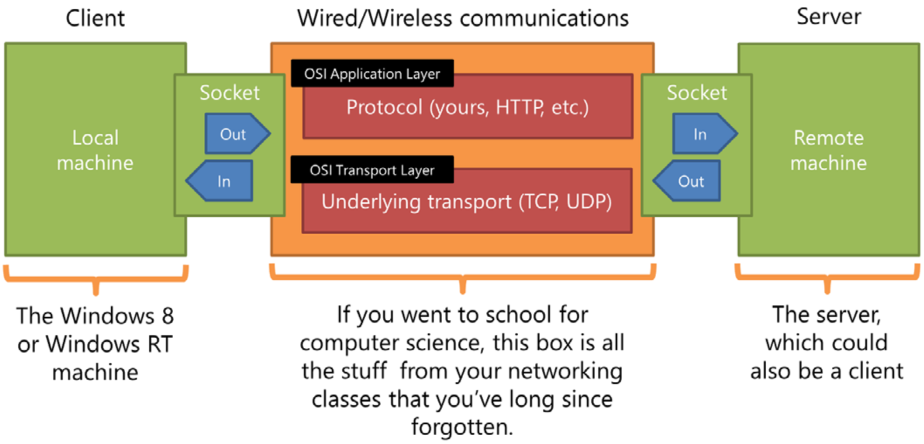

Client
Wired/Wireless communications
Server
Local machine
Socket
Out
In
OSI Application Layer
Protocol (yours, HTTP, etc.)
OSI Transport Layer
Underlying transport (TCP, UDP)
Socket
In
Out
Remote machine
The Windows 8 or Windows RT machine
If you went to school for computer science, this box is all the stuff from your networking classes that you've long since forgotten.
The server, which could also be a client

Figure 18.5 Sockets are the inputs and outputs for each machine. Common protocols like HTTP use sockets underneath. When using socket communication directly, you control the protocol.

connection is established between two endpoints, both sides participate equally in the conversation. It's this listening action that makes something a server.

Figure 18.5 shows how sockets and the endpoints fit into the communications.

Sockets are identified by a service name or, more commonly, a port number. Taking a web server, for example, the server-side socket is typically port 80 (or 443 for secure sockets). The local socket is almost always dynamically allocated from the higher range. So, to make your app a server, you need to listen on the specific socket number, and clients need to know that's the socket number to connect to.

Because the communication between the two endpoints is simply the sockets with whatever protocol or messaging you layer on top of it, the endpoints don't need to be built from the same source code. In fact, in the case of a web server, you have two completely different pieces of software with the browser on the client and the web server on the server. I've also used sockets to control a .NET Micro Framework robot from a Windows 8 app; again, completely different software. For this app, we'll make the app support both the client and server roles for a truly peer-to-peer experience. For any given instance of the app on a machine, only one role is active at a time.

Commercial chat apps typically involve a completely separate server to enable tracking who are online or offline at any given time and what their addresses are. Clients connect to that server initially and may then have direct peer-to-peer connections afterward or simply route everything through the server.

In our app, the chat function will use TCP sockets to enable communication between two Windows 8 PCs but in a purely peer-to-peer way: One PC will establish itself as the server and the other will be able to connect to it. Figure 18.6 shows the flow of the connection.

For simplicity in this example, the IP address (or host name) for the server app will be hardcoded. You'll change that approach later in this chapter when you build out

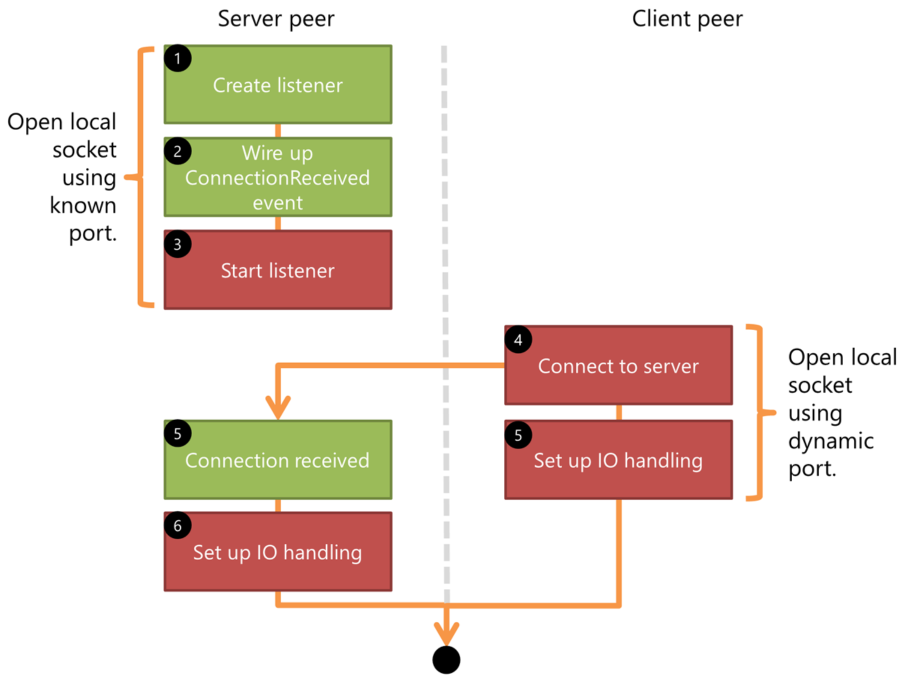

Server peer
Client peer
Open local socket using known port.
1 Create listener
2 Wire up ConnectionReceived event
3 Start listener
4 Connect to server
5 Set up IO handling
Open local socket using dynamic port.
5 Connection received
6 Set up IO handling

Figure 18.6 Chat app connection flow. Step 5 happens logically simultaneously on each machine. Once the connection is established, the peers are on equal footing. The remote port on the server is a known port number. In most cases, you don't control the local port number from code. You can, but typically you'll let the system dynamically allocate that. This is how browser connections work, for example.

the larger app. It's also important to note that once the connection is established, the peers are truly peers—the listener is no longer required because the communication is handled by reading from and writing to streams. In fact, as you'll see shortly, the IO handling code on each machine is identical.

The `MainViewModel` code to listen for a connection is shown here.

Listing 18.8 `MainViewModel` additions for the socket server

```
public async void Listen()
{
  _listener = new StreamSocketListener();
  _listener.ConnectionReceived += OnConnectionReceived;

  await _listener.BindServiceNameAsync(PortOrService);

  var hostNames = NetworkInformation.GetHostNames();

  ConnectionStatus = "Waiting for connection on: ";

  int i = 0;
```

Create listener

Wire up connection handler

Listen on port

```
  foreach (HostName name in hostNames)
  {
    if (i > 0)
      ConnectionStatus += " and ";

    ConnectionStatus += name.DisplayName;
    i++;
  }
}

void OnConnectionReceived(StreamSocketListener sender,
            StreamSocketListenerConnectionReceivedEventArgs args)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    ConnectionStatus = "Connection received from " +
      args.Socket.Information.RemoteHostName.DisplayName;

    _socket = args.Socket;

    SetUpInputHandling();
  });
}
```

Display names for this server

Execute on UI thread

Display connection source

Cache open socket

Set up input handling

This code establishes the running app as a socket server. It first creates a `StreamSocketListener` instance. This class, through the `BindServiceNameAsync` function, is what handles opening up a port and routing communication from that port to your app. When a connection is received on that port, the `ConnectionReceived` event fires.

The code in this method also uses the `DispatcherHelper` class, provided as part of MVVM Light. The `CheckBeginInvokeOnUI` will dispatch the function to the UI if and only if it isn't already running on the UI thread (or, more correctly, has access to the UI thread). If the code is running on the UI thread already, it'll simply execute the code. This method is helpful because the connection received event doesn't fire on the UI thread, but the code (setting the `ConnectionStatus`, for example) requires access to that thread.

Once inside the `ConnectionReceived` handler, all you really need is the socket. The socket passed in is what provides your read/write communications pathway with the remote machine. The code then calls `SetUpInputHandling`, the body of which is shown next.

Listing 18.9 Handing incoming messages in the `MainViewModel` class

```
private DataWriter _writer;
private DataReader _reader;

private void SetUpInputHandling()
{
  _writer = new DataWriter(_socket.OutputStream);
  _reader = new DataReader(_socket.InputStream);

  var t = Task.Factory.StartNew(async () =>
  {
```

Create output writer

Create input reader

Start background thread

```
  _reader.InputStreamOptions = InputStreamOptions.Partial;

  while (true)
  {
   var count = await _reader.LoadAsync(512);

   var message = _reader.ReadString(count);

   DispatcherHelper.CheckBeginInvokeOnUI(() =>
   {
    var chatMessage = new ChatMessage();
    chatMessage.Message = message;

    ChatMessages.Add(chatMessage);
   });
  }
 });
}
```

Return with partial read

Read up to 512 characters

Read string

Add message on UI thread

The `SetUpInputHandling` function first creates the reader and writer for the streams. The `OutputStream` property is where you write data to send it to the other machine. The `InputStream` is where messages received on this machine show up.

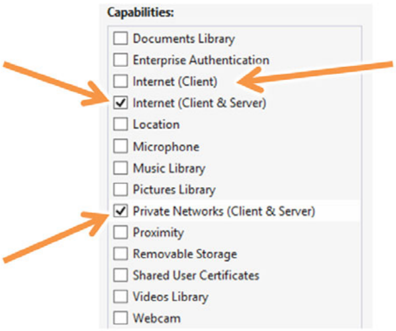
The function is named the way it is because its primary responsibility is to spin up a background thread that monitors in the incoming stream of data.

The background thread, started using the `StartNew` method of the task factory, is an endless loop that waits on the `LoadAsync` method of the `DataReader` class. The `DataReader` (and `DataWriter`) class isn't required for handling IO on the stream, but it certainly makes the task a lot easier by providing high-level methods that understand how to read fundamental types like `int`, `string`, `double` and more.

Finally, in the appx manifest you'll see the Internet Client capability set by default (this was discussed in the previous chapter). For this solution to work, you'll need to also add the Internet (Client and Server) capability. For this app to work on a local network, across machines, you'll need to set the Private Networks (Client & Server)

Select this to use over public networks.

Select this to use over private networks, such as your home or office.



Capabilities:

☐ Documents Library
☐ Enterprise Authentication
☐ Internet (Client)
☑ Internet (Client & Server)
☐ Location
☐ Microphone
☐ Music Library
☐ Pictures Library
☑ Private Networks (Client & Server)
☐ Proximity
☐ Removable Storage
☐ Shared User Certificates
☐ Videos Library
☐ Webcam

This is included in Internet (Client & Server) so it doesn't need to be specified separately.

Leaving it checked is OK, but will bother anyone with OCD on your team.

Figure 18.7 To use this app, you'll need to request server permissions, not just client permissions. Additionally, if this app is used on a private network (this is a domain network or any network with sharing turned on), the Private Networks capability must be selected. Removing the Internet (Client) capability is optional.

capability. Although not necessary, I recommend unchecking the Internet Client capability. Figure 18.7 shows the set of capabilities to request for this app.

With these changes made to the manifest, the server peer is now set up and waiting for connections from the client peer.

18.4 Connecting to the server and sending data

Connecting to an existing server is much simpler than waiting for a connection from a client. Or, at least it's quite a bit simpler in WinRT than it is in Silverlight. I recall the client sockets code for connecting in Silverlight was pages long due primarily to the callback-based async approach. I'm happy this model was greatly simplified for Windows Store apps, because connecting via sockets is an essential task for so many games and communications apps.

Connecting to an existing server is an easy task, primarily handled with a single line of code. Figure 18.8 shows the basics of the workflow, including the additional steps to send data to the server peer.

In this section we'll look at how to open a socket connection to another machine. Once the connection is established, you'll learn how to use the `DataWriter` to write to the socket's input stream.

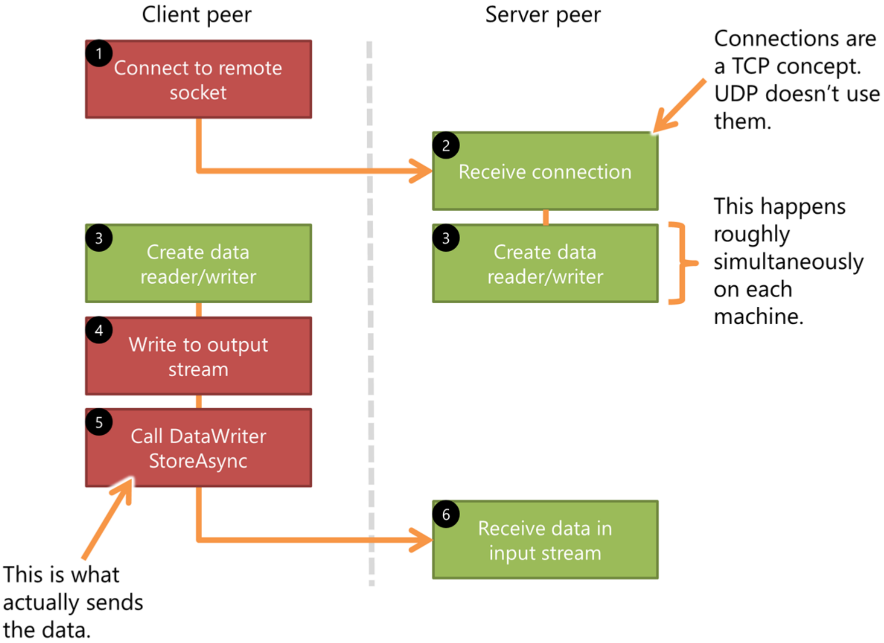

Client peer
Server peer
1 Connect to remote socket
2 Receive connection
3 Create data reader/writer
3 Create data reader/writer
4 Write to output stream
5 Call DataWriter StoreAsync
6 Receive data in input stream
Connections are a TCP concept. UDP doesn't use them.
This happens roughly simultaneously on each machine.
This is what actually sends the data.

Figure 18.8 Data sending workflow. First the client connects to the server, and then (in the case of TCP sockets) both sides create the read/write data streams. The client then writes to the output stream, but the message isn't sent until you call the `StoreAsync` method of the `DataWriter`.

18.4.1 Connecting to an endpoint

In socket communication, the two sides of the conversation are equals. Although it may be convenient to refer to one as the client and the other as the server, they are technically just endpoints. That said, someone has to be listening at first, and someone else has to connect to them. They aren't equals until that connection is established. Think of it like the telephone: Someone had to call, and someone had to be listening for the phone to ring, but once the chatting starts, it doesn't matter who called whom (well, unless the call is an argument about how "you never call." No, I'm not bitter.)

The next listing shows the connect code to place in the `MainViewModel` class.

Listing 18.10 `MainViewModel` code to connect to an existing server

```
public async void Connect()
{
  var hostName = new HostName(ServerAddress);

  _socket = new StreamSocket();

  await _socket.ConnectAsync(hostName, PortOrService);

  SetUpInputHandling();

  ConnectionStatus = "Connected to server at " + ServerAddress + ".";
}
```

Create socket

Connect to server

Set up input handling

This code first creates a `HostName` instance using the `ServerAddress`. `HostName` is a flexible class that can use an IP address, local name, or any other DNS-recognized endpoint name.

Once the `HostName` is created, the code creates a socket. This differs from the server code in that the socket isn't provided to the code in an event handler here; instead you explicitly create the socket and then call `ConnectAsync` passing in the host name and the port number.

After creating the connection, the code calls the same `SetUpInputHandling` code that the server peer code calls. At this point, the two machines are equals and will communicate solely via the streams associated with the sockets.

18.4.2 Sending data

A chat app must be able to send new messages. This is accomplished by writing the message data to the stream. How you structure your message at this point is critical, because you need to know how to parse it on the receiving end, where the data is just raw bytes.

For our first example, we'll keep it simple and post only the string message, as shown in the following listing.

Listing 18.11 `MainViewModel` code to post a new message

```
public async void PostNewMessage()
{
  ChatMessages.Add(NewMessage);

  _writer.WriteString(NewMessage.Message);

  await _writer.StoreAsync();

  CreateNewMessage();
}
```

Add local message

Add message text to stream

Send the message

This code first adds the message to the local collection. This is so you can see your own messages on the message timeline. It then writes the message string to the socket using the `DataWriter` created in the `SetUpInputHandling` method. Writing to the stream isn't enough to send the message, however. To do that, you need to call the `StoreAsync` method of the `DataWriter`. That naming may seem a little odd, but the `DataWriter` isn't something specific to sockets—it could work with file streams where the naming makes a bit more sense.

Finally, the `CreateNewMessage` method creates a new message for the UI to bind to by "newing" one up and assigning `NewMessage` to that new instance.

Run the app on the two different machines. On the server machine, hit the "be a server" button. On the client machine, hit the button to connect to the server. Now, enter a message in either chat `TextBox`. You should see the message echoed locally and (quite quickly) reflected on the screen on the other machine. You've just implemented something that can be the basis for most any multiuser network app or multiplayer game. Sure, the networking graph gets more complex when you add additional players, but the basics of communication are the same.

The code does look a bit sloppy in the `MainViewModel`, though. It's not my style to shove everything in there, but it does make it simpler to learn. Now that you understand how the code works, let's clean it up. I simply couldn't live with myself if we didn't.

18.5 Refactoring for better structure and flexibility

So that you could focus on learning how to use sockets and not worry about architecture, all of the socket communication in the app is currently inside the `MainViewModel`. This made for the least amount of mental overhead when learning the relatively complex topic of socket communications.

As you know from previous chapters, I prefer to factor that type of "guts" code out into separate services classes. This adds a little bit of complexity to the app architecture but provides for a nice clean structure and the ability to expand our approach to include other communications mechanisms. In our case, it's going to provide the ability to support an additional type of communications transport layer, as well as additional messages we'll use in this chapter and the next.

In this new architecture, the viewmodel's public interface will remain the same, and so the UI will remain the same as well (with just one addition). The socket com-

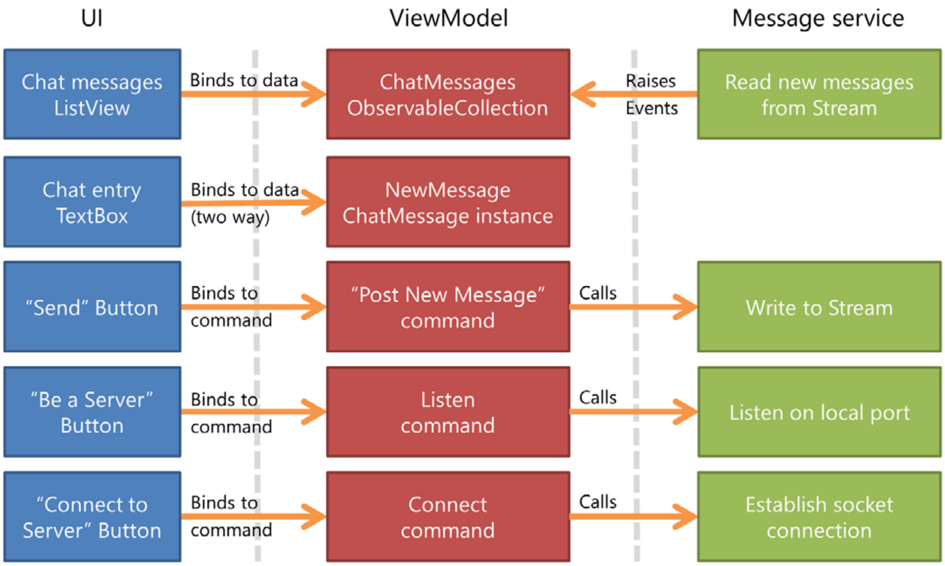

UI
ViewModel
Message service
Chat messages ListView
Binds to data
ChatMessages ObservableCollection
Raises Events
Read new messages from Stream
Chat entry TextBox
Binds to data (two way)
NewMessage ChatMessage instance
"Send" Button
Binds to command
"Post New Message" command
Calls
Write to Stream
"Be a Server" Button
Binds to command
Listen command
Calls
Listen on local port
"Connect to Server" Button
Binds to command
Connect command
Calls
Establish socket connection

Figure 18.9 The refactored app architecture. Except for the addition of player information we'll cover shortly, the public viewmodel interface remains the same, so no changes are required at the UI level.

munication will be factored out into a separate service class. Figure 18.9 shows the changed architecture.

At this level, the architecture looks almost identical to our original. The real difference is the addition of a new class that contains all the sockets code originally in the viewmodel.

In order to support using other socket communications types, such as UDP, you'll also extract an interface that's common across any communications mechanism. Finally, you'll add a few little details that weren't in the initial version, such as providing the Windows username as part of the message information. This will require refactoring and updating the model objects and the viewmodel. To support displaying the name, you'll also make a small update to the main page XAML.

18.5.1 The updated ChatMessage class

The previous version of the `ChatMessage` class had only a single property: the message text. You now want to support the name of the person who sent the message. The name (and potentially other properties in the future) is encapsulated in the `Player` class and surfaced through the `Player` property of the `ChatMessage` class, as shown in the following listing.

Listing 18.12 The updated `ChatMessage` class with new `Player` property

```
using GalaSoft.MvvmLight;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SocketApp.Model
{
  public class ChatMessage : ObservableObject
  {
    private Player _player;
    public Player Player                                  <-- New Player property
    {
      get { return _player; }
      set { Set<Player>(() => Player, ref _player, value); }
    }

    private string _message;
    public string Message
    {
      get { return _message; }
      set { Set<string>(() => Message, ref _message, value); }
    }
  }
}
```

The `Player` class is, like the `ChatMessage` class, a class in the Model folder. Unlike `ChatMessage`, `Player` doesn't inherit from `ObservableObject` because you don't expect to make changes to the object after it is created. The next listing has the new class source.

Listing 18.13 The new `Player` class in the Model folder

```
using System;
using System.Linq;

namespace SocketApp.Model
{
  public class Player
  {
    public string Name { get; set; }                          <-- Player name
  }
}
```

The `ChatMessage` class now exposes a `Player` property, which itself is a `Player` instance with a `Name` property. In order to make use of this new property, you'll need to make a small update to the `DataTemplate` in the `ListView` on MainPage.xaml. Replace the entire `ListView` with the markup from this listing.

Listing 18.14 The updated chat message `ListView`

```
<ListView Grid.Row="0" Margin="0,0,0,10"
          ItemsSource="{Binding ChatMessages}">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Grid Margin="0,5,0,5">
```

```
            <Grid.RowDefinitions>
              <RowDefinition Height="Auto" />
              <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <TextBlock FontSize="24" Grid.Row="0" Margin="2"
                       Foreground="{StaticResource HighlightBrush}"
                       Text="{Binding Player.Name}"
                       TextWrapping="Wrap" />
            <TextBlock FontSize="20" Grid.Row="1" Margin="20,0,0,10"
                       Text="{Binding Message}" TextWrapping="Wrap" />
          </Grid>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
```

Two rows in the grid

Player name in top row

With that, the UI and model object changes are complete. The next step is to create the interface for the messaging service.

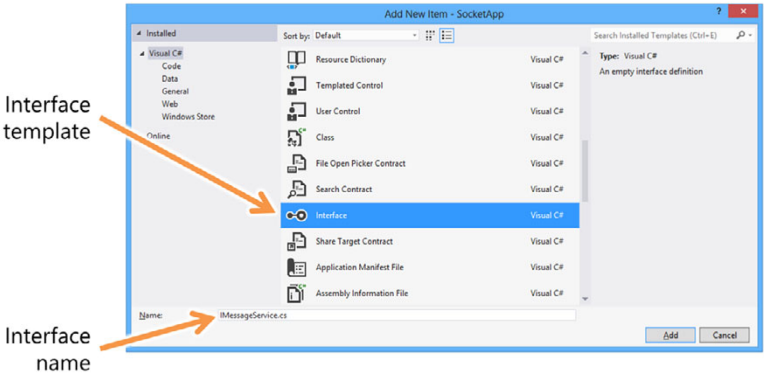
18.5.2 The IMessageService interface

One of the goals of this refactor is to make it easier to swap out different networking implementations. A classic approach for this is to define a common interface and use only that interface from the code. Although you won't use it here, using an interface also opens up the ability to use Dependency Injection (DI) and locator patterns to automatically wire up concrete types. You'll take a simpler approach and create the concrete type in the constructor of the viewmodel.

First, add a new folder named Services in the root of your project. This folder will contain the interface as well as the concrete classes that implement it. As was the case in previous chapters, *Services* in this context means a class that provides functionality to other classes in the app.

Next, add a new interface named `IMessageService`. You could simply create a class and then replace the contents, or you can use the Interface project template shown in figure 18.10.

The `IMessageService` interface exposes the functions that will be used by the viewmodel, specifically, connection and disconnection, listening for new connections,


Add New Item - SocketApp
Installed
Visual C#
Code
Data
General
Web
Windows Store
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Resource Dictionary Visual C#
Templated Control Visual C#
User Control Visual C#
Class Visual C#
File Open Picker Contract Visual C#
Search Contract Visual C#
Interface Visual C#
Share Target Contract Visual C#
Application Manifest File Visual C#
Assembly Information File Visual C#
Add
Cancel
Interface template
Interface name

Type: Visual C#

An empty interface definition

Name: IMessageService.cs

Figure 18.10 Into the Services folder, add a new interface named `IMessageService`.

and sending a message, and then events for the various communications from the service back to the viewmodel. The following listing has the interface source.

Listing 18.15 The `IMessageService` interface

```
using SocketApp.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using Windows.Networking;

namespace SocketApp.Services
{

  // this space intentionally left blank

  public interface IMessageService
  {
    void Connect(Player me, string remoteHostName);
    void Disconnect();
    void Listen(Player me);

    void SendChatMessage(ChatMessage message);

    IReadOnlyList<HostName> GetHostNames();

    event EventHandler<ChatMessageReceivedEventArgs>
                                          ChatMessageReceived;
    event EventHandler<ConnectionReceivedEventArgs> ConnectionReceived;
    event EventHandler<PlayerJoinedEventArgs> PlayerJoined;
    event EventHandler<PlayerExitedEventArgs> PlayerExited;
  }
}
```

Supporting types will go here

Open or close connection

Events

In addition to the expected connection management functions, there are several new event handlers. Two of the event handlers are there to tell you when players join and when they exit. For this chapter, you'll only work with player joining, because player exiting will also require handling things like the app going into suspension—a topic for another chapter.

The event handlers require a number of supporting types, as shown in the following listing. Place this source code in the same file in the spot designated in listing 18.15.

Listing 18.16 Supporting types for the `IMessageService` interface

```
public enum WireMessageType
{
  ChatMessage,
  PlayerJoin,
  PlayerLeave
}

public class ChatMessageReceivedEventArgs : EventArgs
{
  public ChatMessage Message { get; private set; }
```

Message type

```
  public ChatMessageReceivedEventArgs(ChatMessage message)
  {
    Message = message;
  }
}

public class ConnectionReceivedEventArgs : EventArgs
{
  public Player Player { get; private set; }
  public HostName HostName { get; private set; }

  public ConnectionReceivedEventArgs(Player player, HostName hostName)
  {
    Player = player;
    HostName = hostName;
  }
}

public class PlayerJoinedEventArgs : EventArgs                Player joined
{
  public Player Player { get; private set; }

  public PlayerJoinedEventArgs(Player player)
  {
    Player = player;
  }
}

public class PlayerExitedEventArgs : EventArgs                Player exited
{
  public Player Player { get; private set; }

  public PlayerExitedEventArgs(Player player)
  {
    Player = player;
  }
}
```

The `WireMessageType` enum is the interesting part of this listing. It's used as the first element of the message going over the wire (or over the air) to tell the code how the rest of the bytes are to be processed. In this app, you'll support three types of messages, the formats of which are shown in table 18.1.

Table 18.1 The three message types supported in the chat app

Type	Description	Format and byte positions
`ChatMessage`	A free-form chat message sent user to user	0-3: Int32 : `WireMessageType ChatMessage` 4-7: Int32 : String (not byte) length of chat message 8-?: string : The chat message
`PlayerJoin`	Notification that a player has joined the conversation	0-3: Int32 : `WireMessageType PlayerJoin` 4-7: Int32 : String (not byte) length of player name 8-?: string : The player's name
`PlayerLeave`	Notification that a player has left the conversation (not used in this chapter)	0-3: Int32 : `WireMessageType PlayerLeave`

Establishing a solid and flexible messaging pattern for your apps is an important step to implementing communications. In the earlier version in this chapter, I sent only a single string, the chat message, because that was really easy to do. You can see from this table that adding additional message types or additional fields for existing message types requires some real thought.

Now that you understand the message structure, let's implement the class that makes it all happen: the `TcpStreamMessageService` class.

18.5.3 The TcpStreamMessageService class

The `TcpStreamMessageService` class is the meat of communications infrastructure in this app. It is to this class that you refactored most of the functionality that was previously in the `MainViewModel`. In addition, because of the new message types and the tracking of player identity, this version is more complex than what was in the `MainViewModel` previously.

Start with creating a new class named `TcpStreamMessageService` in the Services folder. Replace the contents of that file with the following code.

Listing 18.17 Overall skeleton of the `TcpStreamMessageService` class

```
using GalaSoft.MvvmLight.Threading;
using SocketApp.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;

namespace SocketApp.Services
{
  public class TcpStreamMessageService : IMessageService
  {
    private class PlayerConnection
    {
      public Player LocalPlayer { get; set; }
      public Player RemotePlayer { get; set; }
      public DataWriter Writer { get; set; }
      public DataReader Reader { get; set; }
      public StreamSocket Socket { get; set; }
    }

    public event EventHandler<ChatMessageReceivedEventArgs>
                                       ChatMessageReceived;
    public event EventHandler<ConnectionReceivedEventArgs>
                                       ConnectionReceived;
    public event EventHandler<PlayerJoinedEventArgs> PlayerJoined;
    public event EventHandler<PlayerExitedEventArgs> PlayerExited;
```

Implement interface

Connection information class

Events

```
    private const string PortOrService = "5150";
    private const int MaxMessageSize = 1024;

    private StreamSocketListener _listener;
    private PlayerConnection _connection;

    public async void Connect(Player me, string remoteHostName){}
    private void CloseConnection() {}
    public void Disconnect() {}

    public async void Listen(Player me) {}

    private void ProcessIncomingMessages() { }

    private async void SendPlayerJoinMessage() {}
    private async void SendPlayerLeaveMessage() {}
    public async void SendChatMessage(ChatMessage message) {}

    public IReadOnlyList<HostName> GetHostNames()
    {
      return NetworkInformation.GetHostNames();
    }
  }
}
```

Connection information

Connection management

Listen as server

Parse and process messages

Send messages

This class has all the functions to implement the `IMessageService` interface, as well as some private functions to provide better code structure.

To make it easier for you to expand on this class to support more players, I've encapsulated the connection information into a class-scoped private class named `PlayerConnection`. You keep only an instance of that private class, but to support multiple players, you may want to keep a dictionary or collection of connections.

OPENING AND CLOSING THE CONNECTION

As you suspected, there's more to this class than what's in this first listing, starting with the connection management code shown in the next listing. Add this to the same `TcpStreamMessageService` class.

Listing 18.18 Connection management

```
public async void Connect(Player me, string remoteHostName)
{
  var hostName = new HostName(remoteHostName);

  var pc = new PlayerConnection();
  pc.Socket = new StreamSocket();

  pc.LocalPlayer = me;
  pc.RemotePlayer = null;

  await pc.Socket.ConnectAsync(hostName, PortOrService);

  pc.Reader = new DataReader(pc.Socket.InputStream);
  pc.Writer = new DataWriter(pc.Socket.OutputStream);
```

Set Player information

Connect

Create readers/writers

```
  _connection = pc;

  ProcessIncomingMessages();

  SendPlayerJoinMessage();
}


private void CloseConnection()
{
  if (_connection != null)
  {
    if (_connection.Writer != null)
      _connection.Writer.Dispose();

    if (_connection.Reader != null)
      _connection.Reader.Dispose();

    if (_connection.Socket != null)
      _connection.Socket.Dispose();

    _connection.LocalPlayer = null;
    _connection.RemotePlayer = null;

    _connection = null;
  }
}

public void Disconnect()
{
  SendPlayerLeaveMessage();
  CloseConnection();
}
```

Start read thread

Send introduction message

Close connection

Disconnect

Note that this class doesn't implement `IDisposable`. But because it's keeping instances of classes that do implement `IDisposable`, the containing class should as well.

The code to connect to an existing peer server is similar to what you had in the original version. The main addition here is the sending of an introduction message to the other machine.

LISTENING AS A SERVER

The server listening code is also very similar to the previous version, as shown in the following listing.

Listing 18.19 Listening as a server

```
public async void Listen(Player me)
{
  _listener = new StreamSocketListener();
  _listener.ConnectionReceived += (s, e) =>
    {
```

Create listener

Wire up received handler

```
      var remoteHost = e.Socket.Information.RemoteHostName;

      var pc = new PlayerConnection();

      pc.Socket = e.Socket;
      pc.Reader = new DataReader(pc.Socket.InputStream);
      pc.Writer = new DataWriter(pc.Socket.OutputStream);
      pc.LocalPlayer = me;
      pc.RemotePlayer = null;

      _connection = pc;

      ProcessIncomingMessages();
      SendPlayerJoinMessage();
    };
  await _listener.BindServiceNameAsync(PortOrService);
}
```

Create reader/writer

Start read thread

Send introduction message

Listen on port

Rather than have a separate `ConnectionReceived` event handler as you did in the first version, you put the event handler into an inline lambda expression inside the `Listen` method. Functionally it's almost identical to the original version, with the same addition of sending an introduction message.

SENDING MESSAGES

So, what about that introduction message? This is where you start seeing some new features in this implementation. The next listing includes the functions to send all the supported message types.

Listing 18.20 Sending messages

```
private async void SendPlayerJoinMessage()
{
  if (_connection.Writer != null)
  {
    string playerName = "(unknown)";
    if (_connection.LocalPlayer != null)
      playerName = _connection.LocalPlayer.Name;

    _connection.Writer.WriteInt32((Int32)WireMessageType.PlayerJoin);
    _connection.Writer.WriteInt32((Int32)playerName.Length);
    _connection.Writer.WriteString(playerName);

    await _connection.Writer.StoreAsync();
  }
}

private async void SendPlayerLeaveMessage()
{
  if (_connection.Writer != null)
  {
    _connection.Writer.WriteInt32((Int32)WireMessageType.PlayerLeave);

    await _connection.Writer.StoreAsync();
  }
}
```

Send introduction message

Send goodbye message

```
public async void SendChatMessage(ChatMessage message)
{
  if (_connection.Writer != null)
  {
    _connection.Writer.WriteInt32((Int32)WireMessageType.ChatMessage);
    _connection.Writer.WriteInt32(message.Message.Length);
    _connection.Writer.WriteString(message.Message);

    await _connection.Writer.StoreAsync();
  }
}
```

Send a chat message

Any message that includes a string (or other variable-length data) must also send a count along with the data. This is so the receiving code can properly parse the contents of the message. You could also use C-style string terminators (\0 or 0x00), but I prefer Pascal-style count prefixes (also used by COM and .NET binary serialization). Figure 18.11 shows a comparison of the two string styles.

I went with the Pascal-style approach. Neither is perfect, and each has its own security concerns, especially when it comes to denial-of-service attacks. For example, a missing terminator in a C-style string can be a mess, while an erroneous high length in a Pascal-style string can cause your app to wait forever. The introduction message is important because it's what the endpoints use to share the name of the person connecting. This is done a single time instead of with each message in order to cut down on the amount of data sent across the wire. You could easily extend this to send the bytes of their profile image along with the name, something you definitely wouldn't want to send with every message.

You can now see why the `DataWriter` can be such a huge help. Instead of having to break types down into arrays of bytes, the writer can natively write strings as well as integer types and many others. This makes the code quite a bit simpler to write and understand.

> **TIP** Whenever possible, using the `DataWriter` instead of working with the streams directly will save you a lot of code and time. The `DataWriter` (and related `DataReader`) include methods for writing and reading common types of data without requiring you to worry about the individual bytes the data breaks down into.

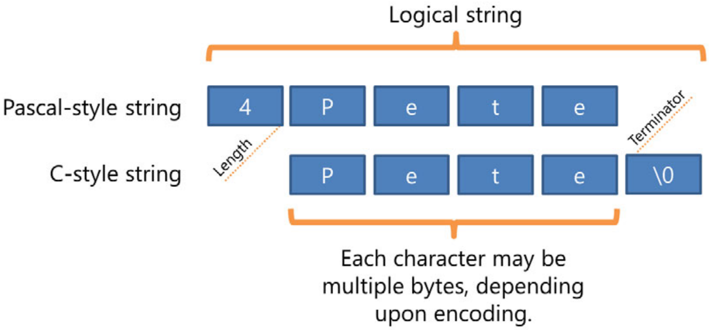


Figure 18.11 The two string styles you may want to consider for your communications.

Other than the additional fields for the length and message type, the process to write and send a message is identical to what you had in the original version: write the data, and then call `StoreAsync`.

PROCESSING MESSAGES

Because of the new message formats, however, reading the messages is more involved. The next listing includes the code for `ProcessIncomingMessages`, which handles all that ugliness.

Listing 18.21 Processing messages

```
private void ProcessIncomingMessages()
{
  var t = Task.Factory.StartNew(async () =>
  {
    _connection.Reader.InputStreamOptions =
                    InputStreamOptions.Partial;

    while (true)
    {
      var count = await _connection.Reader.LoadAsync(MaxMessageSize);

      var messageType = (WireMessageType)_connection.Reader.ReadInt32();

      switch (messageType)
      {
        case WireMessageType.PlayerLeave:
          if (PlayerExited != null)
            PlayerExited(this, new
                  PlayerExitedEventArgs(_connection.RemotePlayer));
          break;

        case WireMessageType.PlayerJoin:
          if (PlayerJoined != null)
          {
            var nameLength = _connection.Reader.ReadInt32();
            var name = _connection.Reader.ReadString((uint)nameLength);

            var remotePlayer = new Player();
            remotePlayer.Name = name;

            _connection.RemotePlayer = remotePlayer;

            PlayerJoined(this, new PlayerJoinedEventArgs(remotePlayer));
          }
          break;

        case WireMessageType.ChatMessage:
          var msgLength = _connection.Reader.ReadInt32();
          var text = _connection.Reader.ReadString((uint)msgLength);

          var msg = new ChatMessage();
          msg.Message = text;
          msg.Player = _connection.RemotePlayer;
```

Start background thread

Return with partial read

Read data

Get message type

Player left

Player joined

Chat message

```
            if (ChatMessageReceived != null)
              ChatMessageReceived(this, new
                           ChatMessageReceivedEventArgs(msg));
            break;
        }
      }
    });
}
```

As in the previous version, this code spins up a background thread that reads from the stream. The `LoadAsync` method is an async call but acts like a blocking call. The execution will halt at that line until there's data to be read.

The `InputStreamOptions.Partial` setting was used in the original version of this code as well. Without this setting, the `LoadAsync` method will return only when it has read `MaxMessageSize` bytes. You instead want it to return when it has read a complete message, regardless of how small it is.

The parsing is the reverse of the writing code. Note how the code reads the string length (in the case of the "player joined" and "chat message" types) and then uses that to tell the reader how many characters to read from the buffer and treat as the string contents.

Message framing

TCP streaming sockets operate on streams of data, not packets of data. Because this is a chat app, where people have to type at a keyboard (or screen), the chances of getting more than one message at once are minimal. The code I've provided here treats it as though you're working with packets of data, even though this isn't technically correct.

But if you're in a situation where you could potentially get a lot more than a single message, you need to loop through all of the unconsumed buffer (using the `UnconsumedBufferLength` property) and read as much as possible, potentially even waiting on more bytes because of partial messages.

In short, with TCP streamed sockets, a single send does not necessarily result in a single receive. TCP isn't doing anything to preserve your message boundaries or frames. You have to do that yourself.

How you go about doing this depends on the size and structure of your messages. I had to do similar processing with MIDI serial communications in the .NET Micro Framework, and trust me, it considerably complicates the parsing code.

Because this class must communicate back with the `MainViewModel`, each of the blocks of code raises an event. Note that because this is executing on a background thread, the events will not be fired on the UI thread. This will be important to know when you get to the `MainViewModel` code.

Finally, you need to update the `MainViewModel` to use this new code. As you can imagine, this will require quite a few changes, because you've essentially gutted it and replaced all of its functionality with this new service.

18.5.4 Updated MainViewModel

The public interface (with the exception of the new `Player` information in the `ChatMessage`) for the `MainViewModel` is the same as what you started with. But there are enough small changes to the code to make me decide to simply provide you with the full source code for the viewmodel, broken across several listings.

INITIAL STRUCTURE

The first of these listings contains the skeleton of the viewmodel as well as the constructor.

Listing 18.22 Overall `MainViewModel` structure

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using GalaSoft.MvvmLight.Threading;
using SocketApp.Model;
using SocketApp.Services;
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;
using Windows.System.UserProfile;

namespace SocketApp.ViewModel
{
  public class MainViewModel : ViewModelBase
  {
    private IMessageService _messageService;

    public MainViewModel()
    {
      _messageService = new TcpStreamMessageService();

      _messageService.ChatMessageReceived +=
                                 OnServiceChatMessageReceived;
      _messageService.ConnectionReceived += OnServiceConnectionReceived;
      _messageService.PlayerExited += OnServicePlayerExited;
      _messageService.PlayerJoined += OnServicePlayerJoined;

      ChatMessages = new ObservableCollection<ChatMessage>();
      ConnectionStatus = "Not connected.";
      ServerAddress = "pete-surface64";

      PostNewMessageCommand = new RelayCommand(
        () => PostNewMessage(),
        () => CanPostNewMessage());

      ListenCommand = new RelayCommand(
        () => Listen(),
        () => CanListen());
```

Service variable

Create service

New event handlers

```
      ConnectCommand = new RelayCommand(
        () => Connect(),
        () => CanConnect());

      InitializePlayer();
      CreateNewMessage();
    }

    private Player _player;
    public Player Player
    {
      get { return _player; }
      set { Set<Player>(() => Player, ref _player, value); }
    }

    private async void InitializePlayer()
    {
      _player = new Player();
      _player.Name = await UserInformation.GetDisplayNameAsync();
    }
  }
}
```

Initialize player

Player

Get Windows username

Notice how in the constructor you create a concrete instance of the `TcpStreamMessageService` class but assign it to a variable of type `IMessageService`. By referring only to the interface in code, you'll need to change only this single line of code in the constructor when it comes time to swap out socket implementations.

The other interesting part of this listing is the code to get the username. The `UserInformation` class has a number of async functions that you can use to get the username, display name, the user's image, and much more.

> **TIP** The `UserInformation` class in the `Windows.System.UserProfile` namespace contains information about the logged-in user. You can use this to get the name, username, and Windows profile picture, among other things.

CONNECTING TO THE SERVER

The next listing has the new code to connect to an existing server. Because of the new service class, the calls here are reduced to just a few lines of code.

Listing 18.23 Connecting as a client

```
private string _serverAddress;
public string ServerAddress
{
  get { return _serverAddress; }
  set { Set<string>(() => ServerAddress, ref _serverAddress, value); }
}

private string _connectionStatus;
public string ConnectionStatus
{
  get { return _connectionStatus; }
  set { Set<string>(() => ConnectionStatus, ref _connectionStatus, value);}
}
```

```
public RelayCommand ConnectCommand { get; private set;}

public void Connect()
{
    _messageService.Connect(Player, ServerAddress);
    ConnectionStatus = "Connected to server at " + ServerAddress + ".";
}

public bool CanConnect()
{
    return true;
}
```

Connect using service

The `Connect` method no longer includes all the code to call out to the sockets functions. Instead, it simply calls the `Connect` function of the message service.

LISTENING FOR CONNECTIONS

Listening for a new connection has been similarly pared down. Here's the code.

Listing 18.24 Listening as a server

```
public RelayCommand ListenCommand { get; private set; }

public void Listen()
{
  _messageService.Listen(Player);

  ConnectionStatus = "Waiting for connection on: ";

  int i = 0;

  foreach (HostName name in _messageService.GetHostNames())
  {
    if (i > 0)
      ConnectionStatus += " and ";

    ConnectionStatus += name.DisplayName;
    i++;
  }
}

public bool CanListen() { return true; }
```

Listen using service

SENDING CHAT MESSAGES

Next, you have the functionality to send chat messages, shown in the following listing. Much of the viewmodel code around chat messages is there just to provide interaction with the UI. The posting of the messages themselves happens in the message service.

Listing 18.25 Sending chat messages

```
public ObservableCollection<ChatMessage> ChatMessages { get; set; }

private ChatMessage _newMessage;
public ChatMessage NewMessage
{
    get { return _newMessage; }
    set { Set<ChatMessage>(() => NewMessage, ref _newMessage, value); }
}

public RelayCommand PostNewMessageCommand { get; private set; }

private void CreateNewMessage()
{
    if (NewMessage != null)
        NewMessage.PropertyChanged -= NewMessage_PropertyChanged;

    NewMessage = new ChatMessage();
    NewMessage.Player = Player;
    NewMessage.PropertyChanged += NewMessage_PropertyChanged;
}
```

Assign player info

```
void NewMessage_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName == "Message")
        PostNewMessageCommand.RaiseCanExecuteChanged();
}

public async void PostNewMessage()
{
    ChatMessages.Add(NewMessage);
    _messageService.SendChatMessage(NewMessage);

    CreateNewMessage();
}
```

Send message using service

```
public bool CanPostNewMessage() { return true; }
```

EVENT HANDLERS

Finally, you have the event handlers. Because the message service exposes a number of events for communicating back to the viewmodel, this is an essential part of the communication. Also, recall how I mentioned that the events aren't coming back on the UI thread: This is handled using the `DispatcherHelper` in each of the event handlers shown in the next listing.

Listing 18.26 Event handlers

```
void OnServicePlayerJoined(object sender, PlayerJoinedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    ConnectionStatus = "Just joined: " +
            e.Player.Name;
  });
}
```

Player joined

```
void OnServicePlayerExited(object sender, PlayerExitedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
```

Player exited

```
    ConnectionStatus = "Just exited: " +
            e.Player.Name;
  });
}
```

New connection

```
void OnServiceConnectionReceived(object sender,
                                 ConnectionReceivedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    ConnectionStatus = "Connection received from " +
            e.HostName.DisplayName;
  });
}
```

New chat message

```
void OnServiceChatMessageReceived(object sender,
                                  ChatMessageReceivedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    ChatMessages.Add(e.Message);
  });
}
```

Each of the event handlers uses the `DispatcherHelper` to ensure the code has access to the UI thread. These handlers are the only communication back to the viewmodel from the message service, so there are handlers for each type of message. If you extend the number of messages your service supports, you'll want to provide appropriate events for those messages as well.

Run the app now, just as you did before. When you connect, you'll see the acknowledgment of the player joining. Once you send a message, you'll notice the additional field in the new data template in the `ListView`.

Those were nice little enhancements thrown in to spice it up a little. The real reason for going through the trouble to refactor and use interfaces is so you can try UDP sockets with very little additional effort.

18.6 Trying out UDP sockets

So far, you've used TCP streaming sockets in this app. Streaming sockets are appropriate for many tasks, but they aren't the absolute lightest weight approach. TCP sockets have additional overhead (both in time and in bytes transferred) for guaranteeing delivery and ordering of messages. In some cases, especially a game sending many real-time updates, it's more critical to be light and fast than it is to worry about getting every single message across the wire.

For those times when you want something a little faster and a little lighter, you have User Datagram Protocol (UDP). UDP is a transport layer protocol like TCP. But unlike TCP, there's no guarantee of delivery or message ordering. If you consider what that means, you can see where something like an audio stream might be better transported over TCP, whereas something like a heartbeat signal or your location in a game at that second might be better transported over UDP. Those aren't hard-and-fast rules, of course. Most audio- and video-streaming protocols build over UDP with all the ordering and other tasks handled higher up in the application layer.

If you want more information on the differences between the two protocols, Wikipedia has two great pages full of details of the headers, reliability, and much more:

- http://en.wikipedia.org/wiki/User_Datagram_Protocol
- http://en.wikipedia.org/wiki/Transmission_Control_Protocol

Choosing a transport protocol isn't something you should take lightly if performance and reliability of communications are important to you. But, with appropriate abstraction in your app, you can avoid locking yourself into any single protocol early in app development.

In this section, you'll build a UDP version of the messaging service for this app. You'll make sure it has the same interface as the TCP version so that either protocol may be easily swapped in or out.

18.6.1 Creating the UdpMessageService class

The UDP implementation of the message service will be in its own class file, just like the TCP version. In the Services folder, add a new class file named `UdpMessageService`, and paste into it the following code.

Listing 18.27 The skeleton for the `UdpMessageService` class

```
using GalaSoft.MvvmLight.Threading;
using SocketApp.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;

namespace SocketApp.Services
{
  public class UdpMessageService : IMessageService
  {
    private class PlayerConnection
    {
      public Player LocalPlayer { get; set; }
      public Player RemotePlayer { get; set; }
      public DataWriter Writer { get; set; }
      public DatagramSocket Socket { get; set; }
    }

    public event EventHandler<ChatMessageReceivedEventArgs>
                 ChatMessageReceived;
    public event EventHandler<ConnectionReceivedEventArgs>
                 ConnectionReceived;
```

Note no DataReader

Identical to TcpStreamMessageService code

```
    public event EventHandler<PlayerJoinedEventArgs> PlayerJoined;
    public event EventHandler<PlayerExitedEventArgs> PlayerExited;

    private const string PortOrService = "5150";
    private const int MaxMessageSize = 1024;
    private PlayerConnection _connection;

    public async void Connect(Player me, string remoteHostName) {}
    private void CloseConnection() {}
    public void Disconnect() {}

    public async void Listen(Player me) {}

    private async void SendPlayerJoinMessage() {}
    private async void SendPlayerLeaveMessage() {}
    public async void SendChatMessage(ChatMessage message) {}
    public IReadOnlyList<HostName> GetHostNames() {}
  }
}
```

Identical to TcpStreamMessage-Service code

Copy full from TcpStreamMessage-Service code

The `SendPlayerJoinMessage`, `SendPlayerLeaveMessage`, `SendChatMessage`, and `GetHostNames` functions are all identical to those in the `TcpStreamMessageService` class, so simply copy them over into this file.

This version of the `PlayerConnection` private class also omits the `DataReader` instance. You'll see why shortly.

18.6.2 Listening for connections

Or, more correctly stated, "binding to a service." The connectivity doesn't work quite the same way it did with TCP streaming sockets. For example, connections aren't persistent and aren't two-way by default. Instead, each side must both connect and listen if it wishes to communicate two-way.

Listing 18.28 Listening for connections

```
public async void Listen(Player me)
{
  _connection = new PlayerConnection();

  BindLocalSocket();
}
private async void BindLocalSocket()
{
  _connection.Socket = new DatagramSocket();
  _connection.Socket.MessageReceived += OnSocketMessageReceived;
  await _connection.Socket.BindServiceNameAsync(PortOrService);
}
```

MessageReceived

Bind

To listen for a connection, you need to create the `DatagramSocket` instance and then wire up the `MessageReceived` event handler. There's no polling loop as you had with TCP, just a discrete message handler. Finally, you bind the socket using `BindServiceNameAsync` or one of the equivalent methods. Doing this establishes a socket you can read from.

18.6.3 Connecting to another machine

The biggest difference between UDP and the TCP code you built earlier is that UDP doesn't have the concept of a persistent connection established before sending data. It just sends data to a specified endpoint. UDP is truly "fire and forget." The `ConnectAsync` function is merely a convenience that handles binding behind the scenes. You could simply bind the port and send messages or even get an output stream with the specified endpoint and start writing.

The following listing establishes the connection to the remote machine, mainly by setting up the data writer and calling the `ConnectAsync` function to handle the binding.

Listing 18.29 Connecting to another machine

```
public async void Connect(Player me, string remoteHostName)
{
  _connection = new PlayerConnection();
  _connection.LocalPlayer = me;

  BindLocalSocket();

  ConnectToRemoteSocket(new HostName(remoteHostName), PortOrService);
}

private async void ConnectToRemoteSocket(HostName hostName,
          string portOrService)
{
  await _connection.Socket.ConnectAsync(hostName, portOrService);
  _connection.Writer = new DataWriter(_connection.Socket.OutputStream);

  SendPlayerJoinMessage();
}

private void CloseConnection()
{
  if (_connection != null)
  {
    if (_connection.Writer != null)
      _connection.Writer.Dispose();

    if (_connection.Socket != null)
      _connection.Socket.Dispose();

    _connection.LocalPlayer = null;
    _connection.RemotePlayer = null;

    _connection = null;
  }
}

public void Disconnect()
{
  SendPlayerLeaveMessage();

  CloseConnection();
}
```

Required to listen for replies

Connect to remote

Set up writer

Send introduction message

Cleanup

Say "good-bye" and clean up

This listing shows more of the two-way dance you need to do when using UDP sockets. In order to be able to send messages, you need to connect. In order to be able to receive messages, you need to bind.

18.6.4 Receiving and parsing messages

Receiving messages is done via an event handler. The event fires only when messages are received. For that reason, there's no need for a loop or the `LoadAsync` method you used with TCP.

The first part of listing 18.30 establishes a writeable connection to the remote socket. This is required because you don't know who is connecting to you up front, and UDP sockets don't offer the convenience of a negotiated connection and the established read/write streams. Instead, if you want to talk to the person who contacted you, you must connect to them.

Listing 18.30 Receiving and parsing messages

```
async void OnSocketMessageReceived(DatagramSocket sender,
                         DatagramSocketMessageReceivedEventArgs args)
{
  if (_connection.Writer == null)
    ConnectToRemoteSocket(args.RemoteAddress, args.RemotePort);

  var reader = args.GetDataReader();
  var messageType = (WireMessageType)reader.ReadInt32();

  switch (messageType)
  {
    case WireMessageType.PlayerLeave:
      if (PlayerExited != null)
        PlayerExited(this,
            new PlayerExitedEventArgs(_connection.RemotePlayer));
    break;

    case WireMessageType.PlayerJoin:
      if (PlayerJoined != null)
      {
        var nameLength = reader.ReadInt32();
        var name = reader.ReadString((uint)nameLength);

        var remotePlayer = new Player();
        remotePlayer.Name = name;

        _connection.RemotePlayer = remotePlayer;

        PlayerJoined(this, new PlayerJoinedEventArgs(remotePlayer));
      }
      break;

    case WireMessageType.ChatMessage:
      var msgLength = reader.ReadInt32();
      var text = reader.ReadString((uint)msgLength);
```

Connect back

```
        var msg = new ChatMessage();
        msg.Message = text;
        msg.Player = _connection.RemotePlayer;

        if (ChatMessageReceived != null)
          ChatMessageReceived(this, new ChatMessageReceivedEventArgs(msg));
        break;    }
    }
```

The `switch` statement that parses the messages is almost identical to the TCP version, except for having to use the `args.GetDataReader` method to get the data reader. Other than that, the method is considerably cleaner because of the lack of looping and data loading. When working with UDP sockets, there exists no persistent stream, so the data reader is provided anew each time data is received.

Finally, crack open the `MainViewModel` and change the constructor so it instantiates the new class:

```
//_messageService = new TcpStreamMessageService();
_messageService = new UdpMessageService();
```

Everything else in the `MainViewModel` class stays the same. If you want to switch back to TCP, simply change this one statement.

If you run the app now, you should see approximately the same thing you saw with the TCP version. My "introduction" message handling isn't identical, but you should be able to send messages back and forth without issue.

For two-way communication, I personally find TCP sockets easier to work with. For very quick one-way communication, however, it's hard to beat UDP sockets. If you get some good core patterns working with UDP, it can be quite efficient to work with. But remember, if you want built-in reliability, you want TCP.

WebSockets

Windows Store apps can also use WebSockets. Despite the name, the protocol doesn't operate over HTTP; only the initial negotiation does. Communication with WebSockets is done over commonly open ports, like port 80, so they are very firewall friendly.

There are two types of WebSockets available in WinRT: the `MessageWebSocket` and the `StreamWebSocket`. The `MessageWebSocket` helps by handling message framing for you but is otherwise conceptually similar to the UDP (Datagram) socket approach. Similarly, the `StreamWebSocket` is conceptually similar to the TCP stream socket we covered earlier.

You typically won't use WebSockets for peer-to-peer communication, because the technology requires a web server for the initial handshake. Instead, you need to set up a proper server (or find one) that's running WebSocket services. For those reasons, I don't go into detail on them here.

For more information, see http://bit.ly/WinRTWebSockets.

18.7 Summary

Socket communication is something that many business developers never run across in their own apps. But in the games, communications, and social network app worlds, it's extremely popular. Sockets provide low-level access without the overhead of a protocol such as HTTP. For that reason, sockets are lightweight and very fast, but for the same reasons, they generally take more effort to use.

In this chapter you built an MVVM app that used both TCP sockets and UDP sockets to communicate between two Windows 8 machines. I was absolutely tickled the first time I realized that a Windows 8 app can be a socket server. That's unusual in a sandboxed environment but enables so many cool scenarios.

TCP sockets are good for reliable communication between two endpoints. I also find them a bit easier to work with compared to UDP when you need to have bidirectional communication. TCP will guarantee delivery and order of data, at the cost of some additional frame overhead and potentially some speed loss.

UDP sockets are good when you want something extremely fast and don't care if it ever gets there. Where TCP is more like FedEx, UDP is more like throwing the package in your cousin's pickup truck and asking him to drop it off. Sure, it'll probably work, and yes, it was a lot cheaper (and maybe even faster), but reliability is definitely a concern.

In the next chapter, we'll expand on the app developed here and update the UI so it works more like a game. We'll add in player graphics, and you'll also learn a bit about Blend and user controls.

19

A little UI work: user controls and Blend

This chapter covers

- Creating a shape in Blend
- Creating and using a `UserControl`
- Creating dependency properties
- Setting view states

In the previous chapter, we created a chat app with the understanding that it would be more game-like in subsequent chapters. The focus was on sockets networking in that chapter, so it made sense not to include graphics and other game functionality. But our app has a giant unused area to the right that's just begging for us to fill it with something cool.

In this chapter, we'll flesh out the game a bit more. Maybe "amusement" is a better term, because this game has no actual gameplay, just movement of elements onscreen in a multiplayer context. (Of course, that makes it better than half the games I downloaded to my older Kindle Fire.)

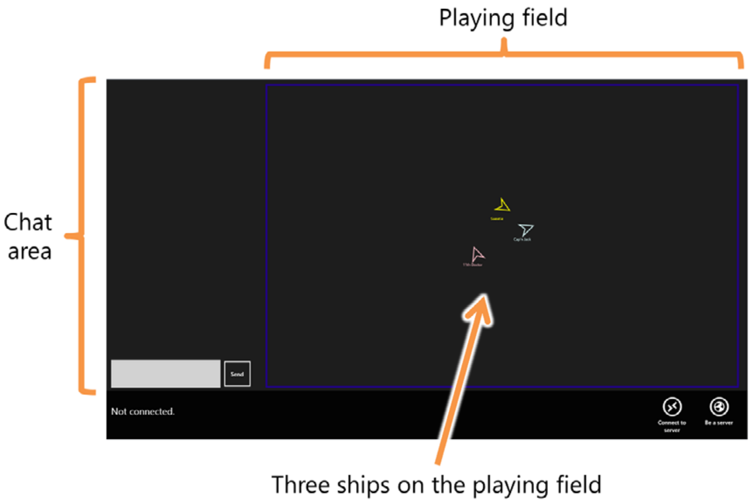


Figure 19.1 The app showing the existing chat area, plus the playing field containing three ships. The ships are hardcoded in XAML for the moment but are implemented as user controls.

Figure 19.1 shows the app as it will appear at the end of this chapter, complete with three copies of the little Asteroids-inspired ship.

The player's ship is a `UserControl` you'll create. It won't yet be wired up to the network, but it will have the UI layer all figured out. First, you'll modify the UI to add a new play area to the right of the chat area. As part of that, you'll also update the page so it properly responds to orientation and view state changes. After that's complete, you'll create a new space ship graphic and user control using Blend and Visual Studio together. This will include information on techniques for binding the user control's UI properties to properties defined in the code-behind and also in the base control class. Once you test the ship by manually placing an instance, or three, of it on the play area, you'll have wrapped up the major UI updates for this app.

19.1 Updated game UI

Our goal in the previous chapters was to get the UI in a state that worked well for the chat demonstration. There was always a large empty area to the right, and the app didn't do anything with either area when it came to changing orientation and filled, snapped, and full view states. We'll address each of those items in this section.

You'll start by making a few important changes to the layout of MainPage.xaml. These changes are required to support the page orientation and view states. Next, you'll create the play field area where the ships will go. You'll wrap up this section with the new visual states to handle the view states and orientation of the page.

19.1.1 Basic changes

In the spirit of refactoring code and markup as you progress through the chapters, there are a few minor changes you need to make to the main page's XAML UI. These are to better support the different screen orientations and states, as well as to hold the playing field. The first listing has the updated MainPage.xaml file, with only the changes. Take note not only of what was added but, as you'll see after the listing, what was removed.

Listing 19.1 The updated MainPage.xaml

```
<common:LayoutAwarePage x:Class="SocketApp.MainPage"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:common="using:SocketApp.Common"
  xmlns:ignore="http://www.ignore.com"
  mc:Ignorable="d ignore"
  DataContext="{Binding Main, Source={StaticResource Locator}}">

  <Page.Resources>
  ...
  </Page.Resources>

  <Page.BottomAppBar>
  ...
  </Page.BottomAppBar>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid Margin="0,0,0,110">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>

      <!-- Chat Area -->
      <Grid Grid.Column="0"
            Width="300"
            x:Name="ChatArea"
            Margin="10,10,10,0">

           ...
      </Grid>

      <!-- Play Area -->

    </Grid>

    <!-- Visual States -->

  </Grid>
</common:LayoutAwarePage>
```

These stay as they were

Now Auto sized

Grid now named

Now fixed sized

The leftmost column, which contains the chat area, has been changed from a fixed size to `Auto` size. This will enable the column to be collapsed when the chat area is collapsed from view. Note also that I removed the third column from the grid. Originally I had intended to use this for navigation controls, but I've decided that direct manipulation of items might be more interesting.

With the column sized to `Auto`, the sizing had to be moved to the `ChatArea Grid` (newly named but always present). This `Grid` has a width of 300 so that, with its margins, it fits within the space allowed a snapped view.

One other very important difference between this listing and the version from the previous chapter is the removal of the `d:DesignWidth` and `d:DesignHeight` properties from the page. These cause problems with the device pane and previewing of the view states while in the designer. With them left in, changing to snapped or filled view or changing orientation doesn't update the rendering of the design surface in a useful way.

19.1.2 Play field area

The play field area is a fixed-size canvas and grid pair contained within a `Viewbox`. Why a `Viewbox`? A `Viewbox` performs pixel scaling of its contents (much like a `ScaleTransform`) to fit the available space. This will enable you to keep the same logical coordinate size for the game area across all machines. If the app enabled a panning game UI with virtual game space, you'd want to eliminate the `Viewbox` to better take advantage of screen real estate on larger displays.

The next listing shows the play area markup to add to MainPage.xaml. Place this in the area designated for the play area.

Listing 19.2 Play area

```
<Grid Grid.Column="1" x:Name="PlayArea"
      Margin="10,10,10,0">
  <Viewbox>                                          ◁ Viewbox
    <Grid Width="1024"                               Natural size
          Height="658">

      <Rectangle StrokeThickness="4"
                 Stroke="{StaticResource AccentBrush}" />     Border

      <Canvas x:Name="PlayField"                     Play field
              Margin="2">
      </Canvas>
    </Grid>
  </Viewbox>
</Grid>
```

For the play field, you use a common technique of putting a `Rectangle` in the background of a `Grid`. You could also use a `Border` element and simply contain the `Canvas` inside that. The `Width` and `Height` properties of the `Grid` are set to the natural size that the `Viewbox` will use when calculating scaling. The size is set to fit within the full view of a 1366 x 768 screen, with a 110 px margin at the bottom for the app bar.

This version of the play area will work well for this chapter, but it's not the final version. When it comes time to add actual ship instances in the next chapter, the structure of this play area will need to change slightly.

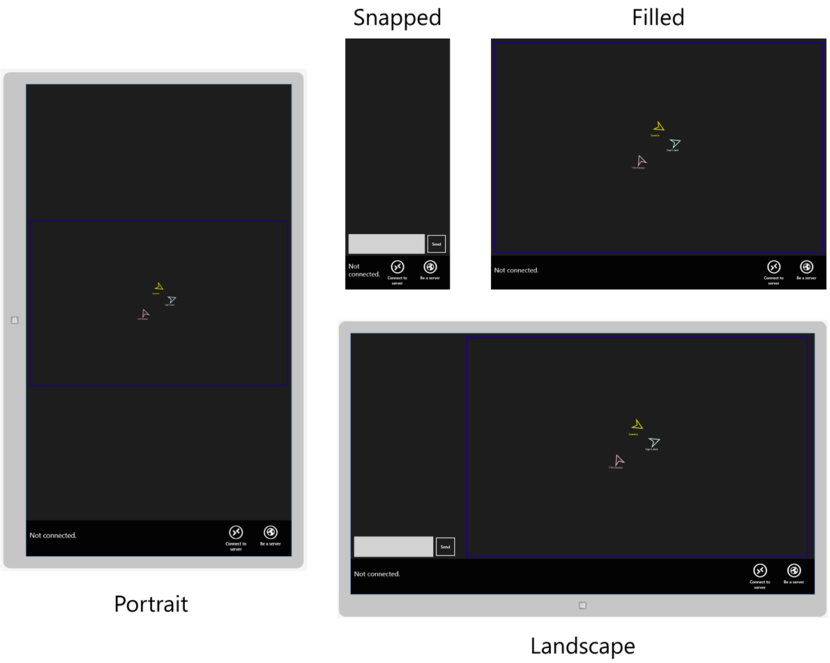

Snapped
Filled
Portrait
Landscape
Not connected.
Connect to server
Be a server
Send

Figure 19.2 The app's view states and orientations, showing the play field and the ship shape you'll eventually place into it.

19.1.3 *Orientation and view states*

Every app needs good support for different orientation and view states. So far, you've lucked out in that the chat area just naturally fits well regardless of those settings. Now, with the addition of new content, you need to provide proper visual states to handle orientation and view state changes. Figure 19.2 shows what we'll be aiming for in this section.

The next listing has the updated visual states for this app. Make the changes to the existing visual states near the bottom of MainPage.xaml.

Listing 19.3 Updated visual states

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ApplicationViewStates">

    <VisualState x:Name="FullScreenLandscape" />

    <VisualState x:Name="Filled">
      <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="ChatArea"
                      Storyboard.TargetProperty="Visibility">
          <DiscreteObjectKeyFrame KeyTime="0"
                                  Value="Collapsed" />
```

Filled view

```
        </ObjectAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>

    <VisualState x:Name="FullScreenPortrait">
      <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="ChatArea"
                      Storyboard.TargetProperty="Visibility">
          <DiscreteObjectKeyFrame KeyTime="0"
                                  Value="Collapsed" />
        </ObjectAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>

    <VisualState x:Name="Snapped">
      <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="PlayArea"
                      Storyboard.TargetProperty="Visibility">
          <DiscreteObjectKeyFrame KeyTime="0"
                                  Value="Collapsed" />
        </ObjectAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Portrait orientation

Snapped view

As a result of these changes, the snapped view will show only the chat area, and both the filled view and portrait orientation views will show only the playing field.

With some more work, you could make it so the chat UI and other elements are all nicely repositioned and resized while always being visible. This may require duplicates of different controls (like using a `GridView` to display chat messages below the play area when in portrait orientation), much as you've done in previous chapters. I'll leave making it that much more awesome to you.

Now that you have a playing area, you need some players to place on it. Next, you'll design the space ship that represents the player.

19.2 Designing the ship UI

Each player in the game will be represented by a small space ship that looks a bit like the classic Asteroids chevron-shaped ship. There are only two players in the game as written (you and your opponent), but I've been designing the interface in such a way as to allow multiple players should you decide to expand on what you've started in this book.

There are some tasks that Blend is simply better suited for when it comes to working with the UI. If you want to create complex animation storyboards, Blend is your tool. If you want to edit control templates, again, Blend is the best choice. If you want to work directly with shapes and paths as you do in this section, Blend is again an excellent choice.

XAML text editor or design surface

I do almost all of my development work directly in Visual Studio, rarely using Blend. I tend to like markup enough that I very rarely use even the built-in design surface in Visual Studio.

That's simply my personal coding style, not a comment on tooling. I got used to hand-typing XAML during the Silverlight and WPF development lifecycles because I was always involved in programs where I got access to private bits early. Invariably, the runtime bits and the designer/Blend bits weren't in sync, so I had to enter the markup as text. I've since carried that habit on to WinRT XAML.

In this section, you'll create a ship shape using Blend for Visual Studio. This ship will be the visual component of the user control you'll build alongside it. As part of this, you'll do a little work with binding to properties in code-behind, defining dependency properties, and creating `UserControl` types. Because I haven't previously spent any meaningful time on Blend in this book, I'll cover the visual design process in detail.

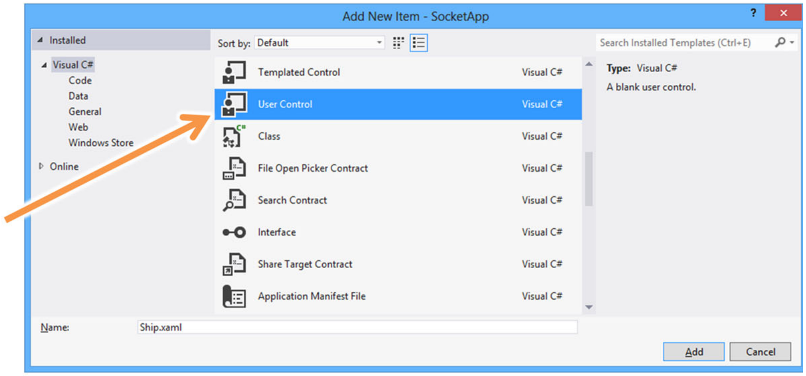
19.2.1 Creating the UserControl

You can create projects and project items in Blend, but I prefer to always start with Visual Studio. That ensures the templates are what I expect them to be, and there aren't any surprises.

First, create a new project folder named Controls. This will be the folder into which the ship user control will be placed.

Next, in the Controls folder, create a new `UserControl` named Ship.xaml, as shown in figure 19.3.

Once the Ship.xaml and Ship.xaml.cs files are in the project, right-click the SocketApp project itself and select the Open in Blend menu option, as shown in figure


Add New Item - SocketApp
Installed
Visual C#
Code
Data
General
Web
Windows Store
Online
Search Installed Templates (Ctrl+E)
Templated Control
User Control
Class
File Open Picker Contract
Search Contract
Interface
Share Target Contract
Application Manifest File
Visual C#
Add
Cancel

Sort by: Default

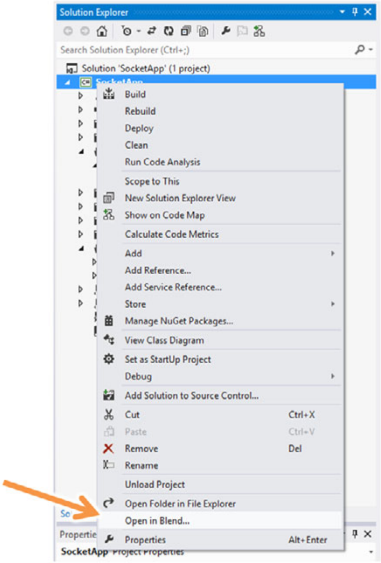
Type: Visual C#

A blank user control.

Name: Ship.xaml

Use the "User Control" project item template.

Figure 19.3 Create the Ship user control using the User Control template in Visual Studio.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'SocketApp' (1 project)
SocketApp
Build
Rebuild
Deploy
Clean
Run Code Analysis
Scope to This
New Solution Explorer View
Show on Code Map
Calculate Code Metrics
Add
Add Reference...
Add Service Reference...
Store
Manage NuGet Packages...
View Class Diagram
Set as StartUp Project
Debug
Add Solution to Source Control...
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Unload Project
Open Folder in File Explorer
Open in Blend...
Properties Alt+Enter
SocketApp Project Properties

Open the project in Blend for Visual Studio 2012.

Figure 19.4 Open the project in Blend to edit the ship's shape.

19.4. Make sure all your files are saved before doing this because this will let you edit everything from within Blend.

TIP Blend for Visual Studio is a free part of the Visual Studio 2012 tool suite. If you don't already have it installed, you can download it from http://dev.windows.com.

If you'd rather not work in Blend, you can skip to the end of this section, where I show the full XAML for the user control.

19.2.2 Creating the ship shape in Blend

With the project open in Blend, you'll start by creating a rectangle. Open the Ship.xaml file in the designer and then add a rectangle using the tool shown in figure 19.5. If your shape tool isn't showing a rectangle (it may show an ellipse or line), click and hold the tool to show the flyout and then select Rectangle from there.

Draw the rectangle directly on the design surface. Make it roughly half the width and height. When you do so, the `Rectangle` element will be added to the main `Grid` of the `UserControl`.

Shape tool showing Rectangle selected

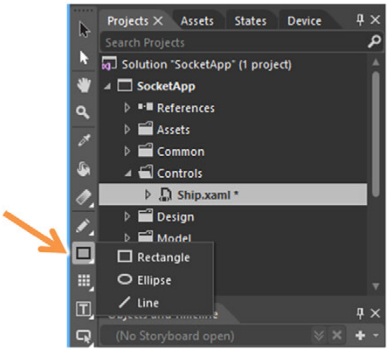
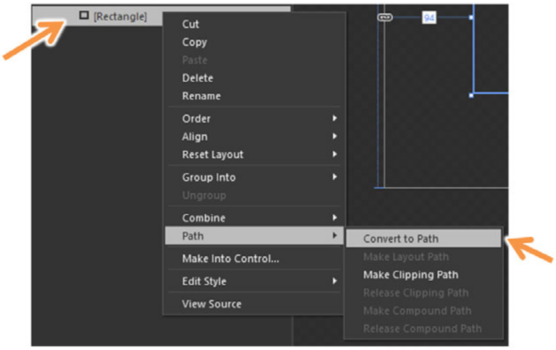

Projects
Assets
States
Device
Search Projects
Solution 'SocketApp' (1 project)
SocketApp
References
Assets
Common
Controls
Ship.xaml *
Design
Model
Rectangle
Ellipse
Line
(No Storyboard open)

Figure 19.5 The shape tool in Blend, showing the Rectangle tool selected. If the Rectangle tool isn't already selected on your toolbar, expand the list of shapes by left-clicking and holding down the shape tool.


[Rectangle]
Cut
Copy
Paste
Delete
Rename
Order
Align
Reset Layout
Group Into
Ungroup
Combine
Path
Make Into Control...
Edit Style
View Source
Convert to Path
Make Layout Path
Make Clipping Path
Release Clipping Path
Make Compound Path
Release Compound Path

Right-click the rectangle in the objects panel.

Convert the rectangle to a path.

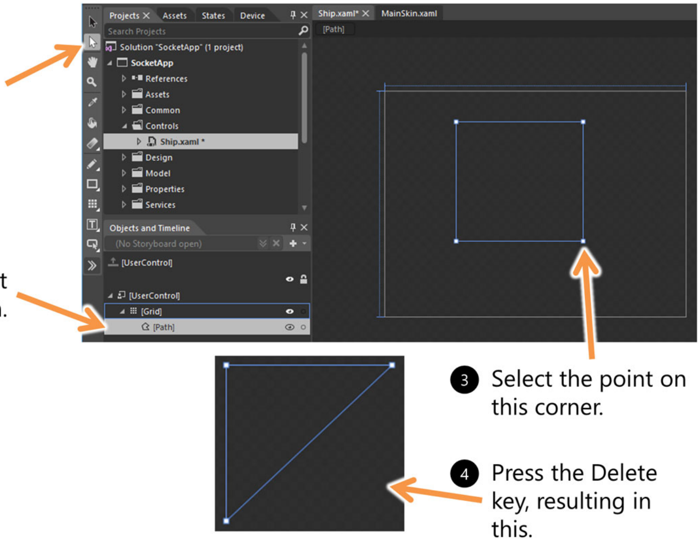
Figure 19.6 How to convert a rectangle shape into a path for point editing

Next, convert the rectangle to a path. Do this by selecting the rectangle either on the design surface or in the objects panel on the bottom left. Right-click it and then choose Path > Convert to Path, as shown in figure 19.6.

Converting the rectangle to a path does exactly what the name suggests: It eliminates the `Rectangle` element and replaces it with a `Path` element made up of the same four corner points.

The reason for this conversion is that you want to change the shape into a triangle. Starting with a rectangle just happens to be the easiest way to do this in Blend.

To change it to a triangle, first select the path, and then choose the point selection tool from the tool palette. This is the white pointer. Next, select the bottom-right corner point using this tool. Finally, with that point selected, hit the Delete key to remove the point. The entire process is illustrated in in figure 19.7.


Projects
Assets
States
Device
Search Projects
Solution 'SocketApp' (1 project)
SocketApp
References
Assets
Common
Controls
Ship.xaml *
Design
Model
Properties
Services
Objects and Timeline
(No Storyboard open)
[UserControl]
[UserControl]
[Grid]
[Path]
Ship.xaml*
MainSkin.xaml
[Path]

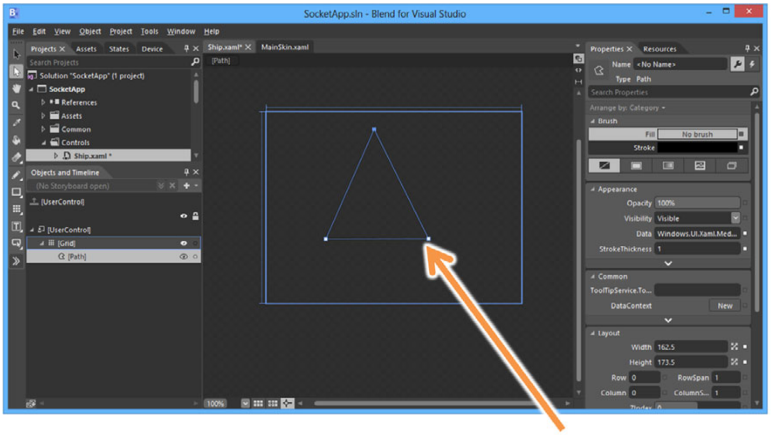
② Choose point selection tool.

① Select path.

③ Select the point on this corner.

④ Press the Delete key, resulting in this.

Figure 19.7 The steps required to easily turn a rectangle into a triangle. (If the rectangle comes up with a white or otherwise nontransparent fill, you can change that using the properties pane to the right. Either reset it or explicitly change to transparent.)


SocketApp.sln - Blend for Visual Studio
File Edit View Object Project Tools Window Help
Projects Assets States Device
Solution 'SocketApp' (1 project)
SocketApp
References
Assets
Common
Controls
Ship.xaml *
Objects and Timeline
(No Storyboard open)
[UserControl]
[Grid]
[Path]
Ship.xaml* MainSkin.xaml
Properties Resources
Name <No Name>
Type Path
Brush
Fill No brush
Stroke
Appearance
Opacity 100%
Visibility Visible
Data Windows.UI.Xaml.Med...
StrokeThickness 1
Common
ToolTipService.To...
DataContext New
Layout
Width 162.5
Height 173.5
Row 0 RowSpan 1
Column 0 ColumnS... 1
100%

Points moved to make a triangle
that looks more like a ship

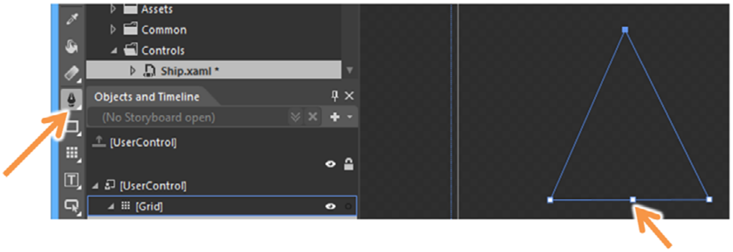
Figure 19.8 Now it's starting to look a little more like a space ship. Well, it does if your judgment criteria are based on 1979 Asteroids.

Now, still using the point selection tool, move the points so they look more like an isosceles triangle, as shown in figure 19.8.

Next, select the pen tool from the toolbar and add a point in the center of the horizontal line on the bottom of the triangle. Like the rectangle tool, the pen tool is grouped with another tool: the pencil tool.

Add the point by hovering over the line until the pen tool shows a small plus sign (+). Figure 19.9 shows the new point added to the triangle.

Pen
tool


Assets
Common
Controls
Ship.xaml *
Objects and Timeline
(No Storyboard open)
[UserControl]
[UserControl]
[Grid]

New point

Figure 19.9 Using the pen tool to add a new point to the bottom of the triangle

Finally, go back to the point selection tool and drag up the new bottom point so you end up with more of a chevron shape. Figure 19.10 shows what you're looking to create.

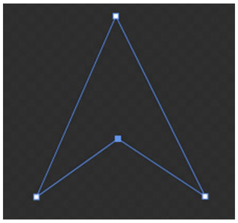


Figure 19.10 The final ship shape

Save the project in Blend. You've finished with Blend for this chapter, so you can close the tool if you'd like. Go back to the project in Visual Studio. If you left Visual Studio open with the `UserControl` source loaded, you'll likely be prompted to reload changes to the Ship.xaml file. Choose Yes at this prompt.

At this point, you can hand-edit the XAML if you want. I'm pretty uptight about fractional sizes and alignment and better with text than a mouse, so I tweaked mine a bit, removed the margin, and sized everything to fit. I also set the `DesignWidth` and `DesignHeight` for the `UserControl`, although those only come into play in the editor. In the Visual Studio editor, the ship now looks like figure 19.11.

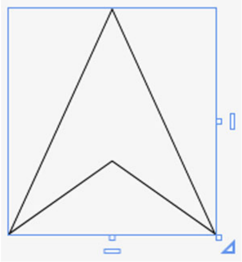


Figure 19.11 The same ship shape in Visual Studio, after tweaking for size and alignment

Finally, I resized it to make the overall shape much smaller and then changed the stroke width to 2 pixels. The updated XAML for the `UserControl` is shown here.

Listing 19.4 The updated ship XAML with everything nicely aligned

```
<UserControl
  x:Class="SocketApp.Controls.Ship"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:SocketApp.Controls"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  d:DesignHeight="30"
  d:DesignWidth="28">

  <Grid>
    <Path Data="M80,-13 L160,160 L80,104 L0,160 z"
          HorizontalAlignment="Left"
          VerticalAlignment="Top"
          Height="30" Width="28"
          Stretch="Fill"
          Stroke="Black"
          StrokeThickness="2"
          UseLayoutRounding="False"/>
  </Grid>
</UserControl>
```

Path data

Scaled size

Stretch to scale

That's it for the main part of the player graphic. You'll expand on this more in a bit by setting colors and such, but first, I've always liked the look of the little IDs next to shapes on systems like airport tower control systems. Let's add a little of that here.

19.2.3 Adding a label

Another part of the player UI will be their name, attached to and floating along with the ship. To do this, you'll add a simple `TextBlock` to the control inside the main `Grid`, under the `Path`. The next listing shows this snippet of markup.

Listing 19.5 The player name `TextBlock` to add after the `Path` in the markup

```
<TextBlock Text="Player Name"
           Foreground="Black"
           HorizontalAlignment="Left"
           VerticalAlignment="Bottom"
           Margin="-15,0,-15,-15"
           TextTrimming="WordEllipsis"
           FontSize="8" />
```

Extend beyond bounds

Truncate if too long

This label will now appear at the bottom left of the ship shape. If the font size is too small for you, experiment with changing both the font size and the margins.

I found Blend perfectly suited for the task of creating this shape, because few people really want to edit paths in a text editor. This was a simple example, of course. More complex graphics are better done in something like Adobe Illustrator and then exported to XAML if you wish to retain their vectors.

That wraps up the visuals for the player. The resulting shape is cool, it's retro, and it was really simple to make. Now you need to make it actually do something. For that, you'll build out properties on the user control.

19.3 Building out the ship user control properties

User controls are really easy to use in apps. They have the same overall structure and model as a page but without the navigation overhead. Typically, one or more user controls are placed on a page to encapsulate functionality that's standalone on the page or reusable across different pages.

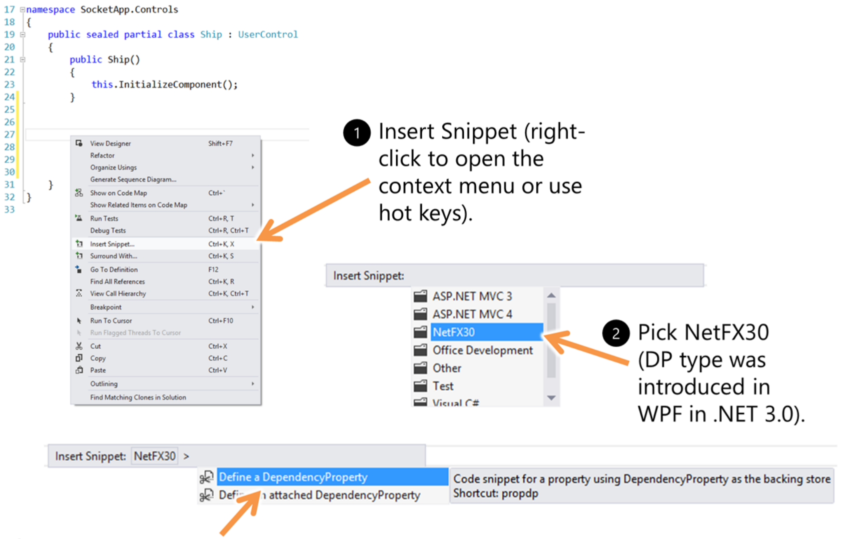
At some point, most every user control needs access to data of some sort. There are two schools of thought when it comes to sharing data with `UserControl`-derived controls:

- *Make the control aware of your model.*

  In this, the control may have a reference to a viewmodel or model objects and do all of its work just like a page. In this way, a user control might natively know how to work with a player or a person.
- *Make the control aware only of discrete properties.*

  This is the approach taken by control vendors and by the framework itself (although they use a slightly different control model from a user control). In this, the user control knows only about discrete properties such as a name or a position.

There's merit to each approach, depending on your app architecture and other considerations. In this section, you'll take the second approach and make the control aware only of discrete properties. You'll add support for rotation as well as for binding the player name and color. Each of the properties you add to the user control will be a dependency property defined in the code-behind and bound to from the user control's markup.



```
namespace SocketApp.Controls
{
    public sealed partial class Ship : UserControl
    {
        public Ship()
        {
            this.InitializeComponent();
        }



    }
}
```

View Designer Shift+F7
Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Show Related Items on Code Map
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, Ctrl+T
Insert Snippet... Ctrl+K, X
Surround With... Ctrl+K, S
Go To Definition F12
Find All References Ctrl+K, R
View Call Hierarchy Ctrl+K, Ctrl+T
Breakpoint
Run To Cursor Ctrl+F10
Run Flagged Threads To Cursor
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Outlining
Find Matching Clones in Solution
Insert Snippet:
ASP.NET MVC 3
ASP.NET MVC 4
NetFX30
Office Development
Other
Test
Visual C#
Insert Snippet: NetFX30 >
Define a DependencyProperty
Define an attached DependencyProperty
Code snippet for a property using DependencyProperty as the backing store
Shortcut: propdp

① Insert Snippet (right-click to open the context menu or use hot keys).

② Pick NetFX30 (DP type was introduced in WPF in .NET 3.0).

③ Pick Define a DependencyProperty.

Figure 19.12 How to define a `DependencyProperty` using the Visual Studio tooling. The first property is named `Angle` and is a `double`. The second is named `PlayerName` and is a `string`. You can also type `propdp` into the code editor and hit the Tab key.

19.3.1 *Enabling rotation*

One thing the ship needs to be able to do is rotate. But you don't want the text to rotate with it. For that reason, the easiest thing to do is to make the rotation act only on the `Path`, not on the `UserControl` as a whole. One excellent way to tackle this is to define an `Angle` dependency property on the user control and allow it to be set by the host of the control.

A quick way to define the property is to use the built-in Visual Studio snippets for dependency properties. These were introduced in order to define dependency properties for WPF 3.0 and above, so they're contained in the NetFX30 folder, as shown in figure 19.12. Following the instructions in this figure, create a dependency property named `Angle` of type `double` in the code-behind of the `Ship` `UserControl`. Then create a `string` dependency property named `PlayerName`.

But if you don't want to create the dependency properties manually, you can paste in the code from the next listing.

Listing 19.6 `Angle` and `PlayerName` dependency properties in `Ship` `UserControl`

```
public double Angle
{
  get { return (double)GetValue(AngleProperty); }
  set { SetValue(AngleProperty, value); }
}
```

Angle property wrapper

```
public static readonly DependencyProperty AngleProperty =
    DependencyProperty.Register(
        "Angle", typeof(double),
        typeof(Ship), new PropertyMetadata(0.0));
```

Angle dependency property

```
public string PlayerName
{
  get { return (string)GetValue(PlayerNameProperty); }
  set { SetValue(PlayerNameProperty, value); }
}
```

PlayerName property wrapper

```
public static readonly DependencyProperty PlayerNameProperty =
    DependencyProperty.Register(
        "PlayerName", typeof(string),
        typeof(Ship), new PropertyMetadata("(unknown)"));
```

PlayerName dependency property

Next, you need to add a `RenderTransform` to enable the `Angle` property to have an impact on the UI. The following listing has the `RotateTransform` (a type of `RenderTransform`) shown in the context of the full markup for the `Ship` `UserControl`.

Listing 19.7 Binding to the `Angle` and `PlayerName` properties

```
<UserControl
  x:Class="SocketApp.Controls.Ship"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:SocketApp.Controls"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  d:DesignHeight="30" d:DesignWidth="28">

  <Grid x:Name="LayoutRoot">
    <Path Data="M80,-13 L160,160 L80,104 L0,160 z"
          HorizontalAlignment="Left"
          VerticalAlignment="Top"
          Height="30" Width="28"
          RenderTransformOrigin="0.5,0.5"
```

Transform origin is center of shape

```
          Stretch="Fill"
          Stroke="Black"
          StrokeThickness="2"
          UseLayoutRounding="False">
      <Path.RenderTransform>
        <RotateTransform Angle="{Binding Angle}" />
```

Angle binding

```
      </Path.RenderTransform>
    </Path>

    <TextBlock Text="{Binding PlayerName}"
```

PlayerName binding

```
               Foreground="Black"
               HorizontalAlignment="Left"
               VerticalAlignment="Bottom"
               Margin="-15,0,-15,-15"
               TextTrimming="WordEllipsis"
               FontSize="8" />
  </Grid>
</UserControl>
```

This listing includes the render transform for rotation but also the binding statements to wire the player name and angle to the code-behind dependency properties. To make this binding work, you need to set the data context for the `LayoutRoot` to be the control itself. Note that you can't use `RelativeSource Self` binding because that won't allow the use of binding outside the control itself (for example, binding the control's `PlayerName` property to a property in the `Player` class). The next listing shows the updated `Ship` control constructor.

Listing 19.8 Updated constructor for the `Ship` control

```
public Ship()
{
    this.InitializeComponent();

    LayoutRoot.DataContext = this;
}
```

DataContext

Note that the `DataContext` for the `UserControl`'s root element is set to the `UserControl` itself. It's important to note that this affects everything inside the root element but allows the `UserControl` itself to have its own data context. Without this distinction, data binding for a `UserControl` would never work correctly because the `UserControl` couldn't have an external data context.

The data context here allows the binding system to find the `Angle` and `PlayerName` properties in the code-behind.

19.3.2 Setting the color

Another thing I wanted to do is to allow each player to have their own color. Rather than define a new dependency property to hold this value, you'll use the `Foreground` property already available for the `UserControl` type. In general, if there's a standard property that can be reasonably used for the property you need, you're better off using it rather than defining a new one.

The next listing shows the changes to the markup to bind to the `Foreground` property.

Listing 19.9 Binding to the `Foreground` property of the `UserControl`

```
<Path Data="M80,-13 L160,160 L80,104 L0,160 z"
      HorizontalAlignment="Left"
      VerticalAlignment="Top"
      Height="30" Width="28"
      RenderTransformOrigin="0.5,0.5"
      Stretch="Fill"
      Stroke="{Binding Foreground}"
      StrokeThickness="2"
      UseLayoutRounding="False">
  <Path.RenderTransform>
    <RotateTransform Angle="{Binding Angle}" />
  </Path.RenderTransform>
</Path>
```

Path stroke

```
<TextBlock Text="{Binding PlayerName}"
           Foreground="{Binding Foreground}"
           HorizontalAlignment="Left"
           VerticalAlignment="Bottom"
           Margin="-15,0,-15,-15"
           TextTrimming="WordEllipsis"
           FontSize="8" />
```

TextBlock foreground

The binding markup follows the same pattern as with the `Angle` and `PlayerName`, even though you're using a dependency property defined by the base class.

This is the last bit of functionality you'll add to the `Ship UserControl`. Other properties, such as its position, will be handled by using attached properties such as `Canvas.Left` and `Canvas.Top` on the entire control.

19.3.3 Temporarily testing the Ship control

This chapter would be a bit of a downer if you didn't get to have a ship visible on the playing field at the end of it. Although you don't yet have the code wired up to handle real players, joined from the network, you can test the UI portion of the playing field and ships by manually adding the `Ship` user control to the markup.

The following listing shows the markup to place one ship control in the `PlayField` canvas. Be sure to add the namespace to the `MainPage` opening tag at the top of the page.

Listing 19.10 `Ship` control to place inside the `PlayField` canvas

```
<common:LayoutAwarePage x:Class="SocketApp.MainPage"
...
        xmlns:controls="using:SocketApp.Controls"
...
>


...
        <controls:Ship Angle="120"
                       PlayerName="Pete Brown"
                       Canvas.Left="500"
                       Canvas.Top="250"
                       Foreground="Yellow" />
...
```

Define controls namespace

Place ship inside PlayField canvas

You can add as many `Ship` controls as you want, each with different rotation angles, positions, names, and foreground colors. You can see in figure 19.13 that I added three ships.

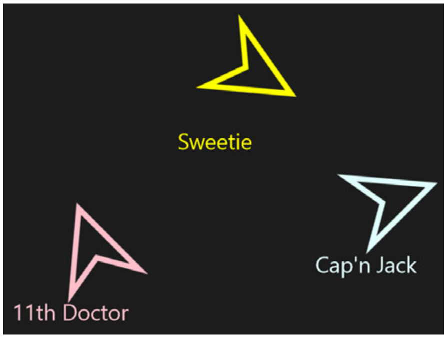


Figure 19.13 Three ships passing in the night (zoomed-in view snagged from the Visual Studio design surface)

Because of the use of dependency properties and binding, the `Angle` and `PlayerName` are easily used from XAML and can be updated from code, with the changes immediately reflected in the presentation.

Once you've completed testing by hardcoding the ships, remove them all so the `Canvas` element contains no children. You'll work on programmatically adding these elements in the next chapter.

19.4 Summary

I'm a big fan of Asteroids and, luckily, their ship is super easy to draw using vector graphics. The Asteroids game itself used vector graphics on a classic tube display—there were no pixels in the game and no raster lines, so we're following the lead from the classic game.

Designing this simple shape in Blend was really easy. You started with a square, removed a corner to get a triangle, moved some points around and then added a final point. If only all graphics design work were this easy! Blend made this easy because Blend is geared toward exactly this type of work: lightweight graphics work, plus animation and design.

By encapsulating the ship in a user control, you were able to keep its interface with the outside world simple: just a few properties. The app's main page and the `PlayField` container don't need to know anything about how the `Ship` is implemented internally. Having it in a user control also made it easy to add a few instances to the page to test it all out.

You've seen dependency properties before, but it's always nice to both learn a different way to create them and to see a practical use of them in action. You defined dependency properties for the `PlayerName` and the `Angle`. You also used an existing dependency property for the foreground brush of the control.

In the next chapter, you'll update the model and services to support the UI work you completed in this chapter.

20

Networking player location

This chapter covers

- The updated `Player` model
- Using an `ItemsControl` to display ships
- Sending player updates across TCP

In the previous chapter you created the `Ship` user control and displayed it on the screen. In this chapter you'll expand on that work. The primary purpose of this chapter is to show you how I filled out the glue code that makes the socket service and player display from the previous chapters respond to the user input from the next chapter. As part of this, you'll learn some interesting techniques such as using a bound `ItemsControl` to display the players on the playing field and spinning up background threads to send test data across the wire.

You'll start by updating the `Player` model class. If you want to send meaningful player information across the network, you'll need more than just a name. Next, you'll create a collection of `Player`s in the game and expose that via the viewmodel. This collection will be maintained via network events and will be the source of data for an `ItemsControl` in the UI. The `ItemsControl` is an interesting use of a UI element because it won't display a typical list of data but will instead render the ships onscreen at the correct positions.

Let's start by expanding the `Player` model object.

20.1 Updating the Player model

The current `Player` class includes only the player's name. To flesh out this game demo, you'll need to expand that to include location information as well as the color assigned to the player's ship.

In this section, you'll first create a new class that encapsulates the player's location and angle information. Then, you'll expose an instance of that from the `Player` class, alongside the property to control the ship's color.

20.1.1 The PlayerLocation class

The `Player` class aggregates an instance of the `PlayerLocation` class. This class describes the position and orientation of the player's ship. The UI will bind to this information when positioning the ship. Of particular interest is the `Angle` property. In the previous chapter, you saw the `Angle` property in action in the `Ship` user control. The `Angle` in the `PlayerLocation` class is a `double`, which represents the full angle starting at zero degrees, as shown in figure 20.1.

Unlike the `X` and `Y` properties, which specify the location of the top-left corner of the ship, the render transform origin specifies that the rotation will happen around the geometric center of the shape. This is due to the render transform origin setting in the `Ship` control.

The `PlayerLocation` has three properties, as shown here. Create a new class file in the Model folder and replace the contents with this code.

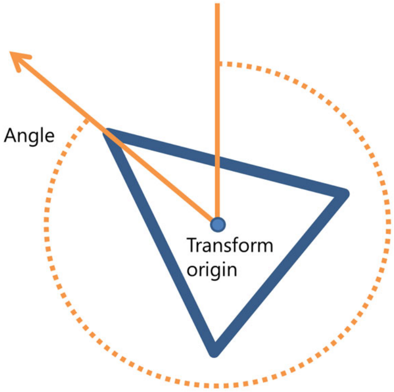


Figure 20.1 The angle is the total angle starting from zero. It can be greater than 360 degrees (370 is equivalent to 10), or it can even be a negative number (-90 is equivalent to +270). WinRT XAML will figure out the correct render transform angle regardless of the format.

Listing 20.1 The `PlayerLocation` model class

```
using System;
using GalaSoft.MvvmLight;

namespace SocketApp.Model
{
  public class PlayerLocation : ObservableObject
  {
    private double _x;
    public double X
    {
      get { return _x; }
      set { Set<double>(() => X, ref _x, value); }
    }

    private double _y;
    public double Y
    {
      get { return _y; }
      set { Set<double>(() => Y, ref _y, value); }
    }

    private double _angle;
    public double Angle
    {
      get { return _angle; }
      set { Set<double>(() => Angle, ref _angle, value); }
    }
  }
}
```

X (left)

Y (top)

Angle

These three properties will be bound to the `Canvas.Left` (`X` property), `Canvas.Top` (`Y` property), and the `Angle` property of the `Ship` control.

20.1.2 The updated Player class

The changes to the `Player` class are small but important. First, you need to hold the color for the ship assigned to this `Player`. To keep the binding simple, this property is implemented as a `SolidColorBrush`. You could make this a straight `Brush` for more flexibility or a simple platform-agnostic RGB value if you want slightly more complex binding.

The second change is the addition of the `Location` property. This property, of type `PlayerLocation`, is how you'll expose the player's location. When looking at the listing, note the property setter for the `Location` property. More details on that in a moment.

The next listing shows the updated `Player` model class, including the new `Location` property.

Listing 20.2 The updated `Player` model class

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Threading;
using System;
using System.Linq;
using Windows.UI.Xaml.Media;

namespace SocketApp.Model
{
  public class Player : ObservableObject
  {
    private string _name;
    public string Name
    {
      get { return _name; }
      set { Set<string>(() => Name, ref _name, value); }
    }

    private SolidColorBrush _color;
    public SolidColorBrush Color
    {
      get { return _color; }
      set { Set<SolidColorBrush>(() => Color, ref _color, value); }
    }

    private PlayerLocation _location = new PlayerLocation();
    public PlayerLocation Location
    {
      get { return _location; }
      set
      {
        DispatcherHelper.CheckBeginInvokeOnUI(() =>
          {
            Set<PlayerLocation>(() => Location, ref _location, value);
          });
      }
    }
  }
}
```

Ship's color

Location

When you expose the `PlayerLocation` instance from the `Player` class, you need to take an extra step and check to see if you have access to the UI thread. Why? Because this property is often set from a background thread, and this is the most convenient place to make that check. Access to the UI thread is needed when setting properties that raise property change notifications to UI elements. As in previous examples, you use the `DispatcherHelper` from Laurent's MVVM Light library.

The changes to the `Player` class and the addition of the new `PlayerLocation` class are all the changes you'll make to the model objects. Together, these now have everything you need to know about a player in the game.

20.2 The collection of players

The viewmodel is the interface between the UI and the model. In order to make the players accessible to the UI, they must be exposed by the viewmodel in a collection. You've learned previously that the most binder-friendly type of collection is the `ObservableCollection`, so that's what you'll use here. The players in the viewmodel will be 1:1 with the ships on the page.

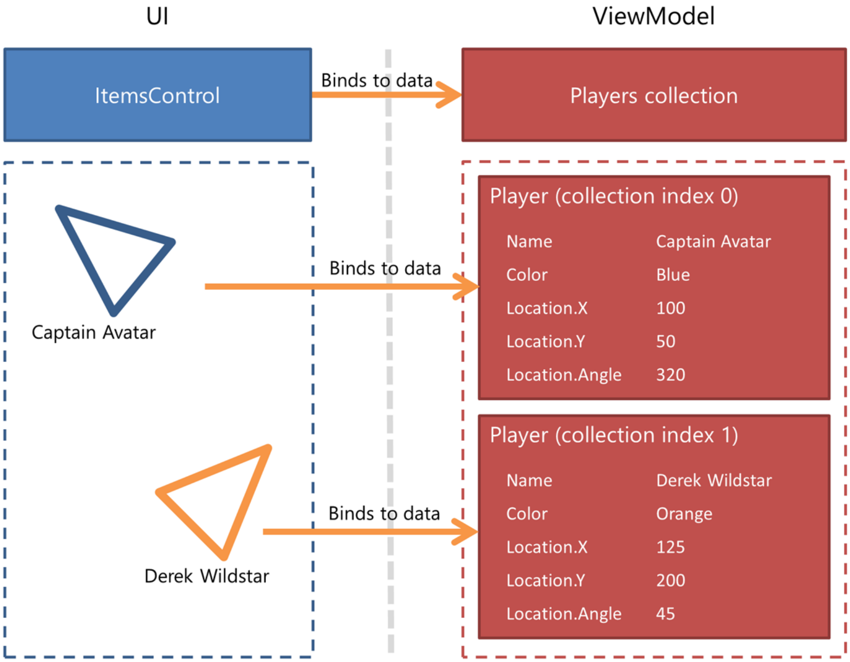

UI
ViewModel
ItemsControl
Binds to data
Players collection
Player (collection index 0)
Name
Captain Avatar
Color
Blue
Location.X
100
Location.Y
50
Location.Angle
320
Captain Avatar
Binds to data
Player (collection index 1)
Name
Derek Wildstar
Color
Orange
Location.X
125
Location.Y
200
Location.Angle
45
Derek Wildstar
Binds to data

Figure 20.2 The `ItemsControl` in the UI displays the ships by binding to the `Players` collection in the viewmodel. Each ship in the UI has a corresponding player in the viewmodel.

This relationship between the viewmodel players and the ships is shown in figure 20.2.

You might think "`ItemsControl` for the UI…really?" (Bonus points if you made the "really?" face when you thought that.) Yes, that's exactly what I've done here. It just goes to show the flexibility of the control model in XAML.

In this section, you'll first set up the `Players` collection and the code that initializes the player instance. Then, you'll update the UI to use an `ItemsControl` in an unusual way: as the playing field. Before wiring up the viewmodel to the network service, you'll do a little testing to ensure the UI can properly reflect the contents of the `Players` collection. We'll wrap up this section with updated event handlers to add players to and remove players from the collection.

20.2.1 Initializing the collection

The `Players` collection contains a list of all the players in the game. For you, this will always be two players, but you could expand the game's capabilities to support more without requiring any changes to this collection or to the UI binding. The code to initialize the collection also creates the local `Player` instance for the person using this instance of the app. This same code also provides the color for this player and later, all other players.

The following listing contains the code to modify in `MainViewModel`.

Listing 20.3 The player-related properties and functions in `MainViewModel`

```
private Random _random;

private Player _player;
public Player Player
{
  get { return _player; }
  set { Set<Player>(() => Player, ref _player, value); }
}

public ObservableCollection<Player> Players { get; set; }

private async void InitializePlayer()
{
  Players = new ObservableCollection<Player>();

  Player = new Player();
  Player.Name = await UserInformation.GetDisplayNameAsync();
  Player.Color = GetNextPlayerColor();

  Player.Location.X = _random.Next(300) + 150;
  Player.Location.Y = _random.Next(200) + 75;
  Player.Location.Angle = _random.Next(359);

  Players.Add(_player);
}

private List<Color> _colors = new List<Color>()
  { Colors.Red, Colors.Green, Colors.Blue,
    Colors.Yellow, Colors.White, Colors.Violet,
    Colors.Azure, Colors.CornflowerBlue,
    Colors.Cyan, Colors.DeepPink, Colors.LightGray};

private SolidColorBrush GetNextPlayerColor()
{
  Color c;

  if (_colors.Count > 0)
  {
    c = _colors[_random.Next(_colors.Count - 1)];
    _colors.Remove(c);
  }
  else
  {
    c = Colors.Wheat;
  }

  return new SolidColorBrush(c);
}
```

Get player name

Set player color

Set random location

Add player to collection

Initialize possible colors

Pick random color

Remove that color

This code creates the initial local player and assigns it both to the `Player` property and to the first element in the `Players` collection. The `Player` property is for convenience—you could simply use `Players[0]` in binding statements in your own app if you prefer.

The color management is another interesting part of this listing. I have a stock set of colors from which the app can choose. I did this because random color generation, without some decent algorithms, almost always generates muddy or pastel colors. To ensure no two ships have the same color, the code removes "used" colors from the list.

Note that this code won't run without error because you haven't yet initialized the `_random` member variable. That will come shortly.

20.2.2 Displaying players with an ItemsControl

In the previous chapter, you manually added the `Ship` user control instances to the `Canvas` on MainPage.xaml. You could perform the same action dynamically in the code-behind, creating the user control instances and then adding them to the `Canvas`. I decided to do something a little more interesting, however. In the next listing you can see that I used an `ItemsControl` and bound it to the `Players` collection of the `MainViewModel`. The `DataTemplate` for the `ItemsControl` displays the `Ship` instance, binding its properties to those on the `Player` it's bound to.

Listing 20.4 Updated playing field area in MainPage.xaml

```
<!-- Play Area -->
<Grid Grid.Column="1"
      x:Name="PlayArea"
      Margin="10,10,10,0">
  <Viewbox>
    <Grid Width="1024" Height="658">
      <Rectangle StrokeThickness="4"
                 Stroke="{StaticResource AccentBrush}" />

      <ItemsControl ItemsSource="{Binding Players}"
                    HorizontalAlignment="Stretch"
                    VerticalAlignment="Stretch"
                    Margin="2">
        <ItemsControl.ItemsPanel>
          <ItemsPanelTemplate>
            <Canvas />
          </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
        <ItemsControl.ItemTemplate>
          <DataTemplate>
            <Canvas>
              <controls:Ship Angle="{Binding Location.Angle}"
                             PlayerName="{Binding Name}"
                             Canvas.Left="{Binding Location.X}"
                             Canvas.Top="{Binding Location.Y}"
                             Foreground="{Binding Color}"
                             Width="28" Height="30" />
```

ItemsControl

ItemsPanelTemplate

ItemTemplate

Ship

```
            </Canvas>
          </DataTemplate>
        </ItemsControl.ItemTemplate>
      </ItemsControl>
    </Grid>
  </Viewbox>
</Grid>
```

Back when I first started using Silverlight, I quickly discovered the versatility of the `ItemsControl`. When I needed to display a number of items on the page but didn't necessarily want a list layout or selection capabilities, I would turn to it as my go-to control. The `ItemsControl` in WinRT XAML is just as flexible and useful.

The `ItemsControl` is the most basic list control, having no real display representation. It simply renders a number of items using the provided panel and item templates. By providing the `Canvas` as the `ItemsPanelTemplate` rather than a `StackPanel`, you can use X/Y coordinates to position the items. This moves you out of the straight stacked-list metaphor and into a free-form layout approach.

Consider how you could use this technique in your own apps: Graphs and charts come to mind as a quick example.

20.2.3 Testing the collection

Before you wire the `Players` collection to the network code, you'll want to test to ensure the user interface is properly displaying all of the players in the collection. There are a number of ways to do this, but I find the easiest is to add a new method to the `MainViewModel`.

The following listing includes the new method that adds a number of random players to the collection.

Listing 20.5 Testing adding players in `MainViewModel`

```
public void TestAddingPlayers()
{
  for (int i = 0; i < 5; i++)
  {
    var p = new Player();
    p.Color = GetNextPlayerColor();

    p.Location.X = _random.Next(300) + 150;
    p.Location.Y = _random.Next(200) + 75;
    p.Location.Angle = _random.Next(359);
    p.Name = "Test Player " + i;

    Players.Add(p);
  }
}
```

Generate five players

Generate color

Set random location

Add to collection

When this code runs, you'll have six players in the game: you and the five generated players.

The `TestAddingPlayers` method needs to be called from somewhere. One easy location is as the last line of the `MainViewModel` constructor. The next listing has this addition.

Listing 20.6 Updated `MainViewModel` constructor

```
public MainViewModel()
{
  _random = new Random();

  _messageService = new TcpStreamMessageService();
  //_messageService = new UdpMessageService();

  ...

  InitializePlayer();
  CreateNewMessage();

  TestAddingPlayers();
}
```

Initialize _random

Make sure you're using the right service

Keep everything else

Test adding players

At this point, you can run the app to test out the new functionality. You should see the app with six ships in random locations, very similar to what you saw in the previous chapter.

Once you've verified that the collection is accurately represented in the UI, comment out the `TestAddingPlayers` call in the constructor; you won't need it beyond this point, and it will be in the way. What you need to do now is wire up the collection management to the events raised from the messaging service.

20.2.4 Wiring up the collection to service events

The `Players` collection works with the test data, so it now needs to be wired up to the message service. Specifically, when a new player joins, they need to be added to the collection. When a player exits, they need to be removed.

The next listing includes the `MainViewModel` code for the `OnServicePlayerJoined` and `OnServicePlayerExited` event handlers. These methods already exist in the class but simply update the `ConnectionStatus` property.

Listing 20.7 Adding and removing players based on network events

```
void OnServicePlayerJoined(object sender, PlayerJoinedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    var p = e.Player;
    p.Color = GetNextPlayerColor();
    Players.Add(e.Player);

    ConnectionStatus = "Just joined: " +
            e.Player.Name;
  });
```

Add player

```
  TestPositionUpdate();
}

void OnServicePlayerExited(object sender, PlayerExitedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    Players.Remove(e.Player);                         Remove player

    ConnectionStatus = "Just exited: " +
            e.Player.Name;
  });
}
                                                          Stub method
private void TestPositionUpdate() {}
```

That wraps up the `Player`-handling code in the UI. If you haven't already, be sure to comment out the call to `TestAddingPlayers` in the constructor.

You have the event handlers set up to manage the viewmodel collection. Now you need to update the TCP streaming message service to send and receive the player location information.

20.3 Updating the TCP stream message service

The UI and viewmodel now know about player location. But the TCP streaming message service is neither sending nor receiving this information as part of its data stream. Without that, you're dealing with only local changes, which makes for a pretty lame network game.

In this section, you'll first define a new message type: `PlayerLocation`. As a part of that, you'll need to define a new event in the interface and the associated `EventArgs` class to go with it. You'll also need a method that sends updated player information to the remote machine. All of this will require that you settle on a format for this data so that you know how to read and write it. Finally, the function that processes all incoming messages needs to be updated to be able to process this specific message type.

20.3.1 Updated message service interface

The `IMessageService` interface defines what functions and events are available on the message services. To this interface, you need to add in the ability to report location. On the sending side, this means a new method the viewmodel can call to update the location. On the receiving side, it means a new event to report the remote player's location.

The next listing includes the updated interface, the `WireMessageType` it uses, and the new `EventArgs` class required for the event.

Listing 20.8 Updated IMessageService.cs entries

```
public enum WireMessageType
{
  ChatMessage,
  PlayerJoin,
  PlayerLeave,
  PlayerLocation
}

...

public class PlayerLocationUpdatedEventArgs : EventArgs
{
  public Player Player { get; private set; }

  public PlayerLocationUpdatedEventArgs(Player player)
  {
    Player = player;
  }
}

public interface IMessageService
{
  void Connect(Player me, string remoteHostName);
  void Disconnect();
  void Listen(Player me);

  void SendChatMessage(ChatMessage message);
  void SendLocationUpdate();

  IReadOnlyList<HostName> GetHostNames();

  event EventHandler<ChatMessageReceivedEventArgs> ChatMessageReceived;
  event EventHandler<ConnectionReceivedEventArgs> ConnectionReceived;
  event EventHandler<PlayerJoinedEventArgs> PlayerJoined;
  event EventHandler<PlayerExitedEventArgs> PlayerExited;

  event EventHandler<PlayerLocationUpdatedEventArgs>
                              PlayerLocationUpdated;
}
```

PlayerLocation

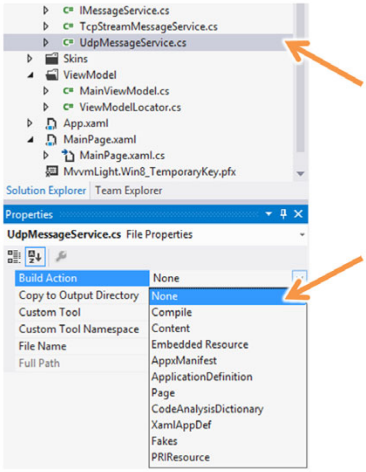
New event args

New method

New event

The changes to the interface follow the same patterns as the original definition: Each message requires an entry in the `WireMessageType` enum and an event to notify listeners when that message is received. On the sending side, the message requires a function to send the message.

I'm not going to take the UDP message service any further, so rather than deal with the compile errors that will come from this updated service definition, *I simply changed the UdpMessageService.cs file's properties so the build action is set to None.* You could also exclude it


IMessageService.cs
TcpStreamMessageService.cs
UdpMessageService.cs
Skins
ViewModel
MainViewModel.cs
ViewModelLocator.cs
App.xaml
MainPage.xaml
MainPage.xaml.cs
MvvmLight.Win8_TemporaryKey.pfx
Solution Explorer
Team Explorer
Properties
Copy to Output Directory
Custom Tool
Custom Tool Namespace
File Name
Full Path
None
Compile
Content
Embedded Resource
AppxManifest
ApplicationDefinition
Page
CodeAnalysisDictionary
XamlAppDef
Fakes
PRIResource

UdpMessageService.cs File Properties

Build Action None

Select the UdpMessageService source file.

Ensure the build action is set to None.

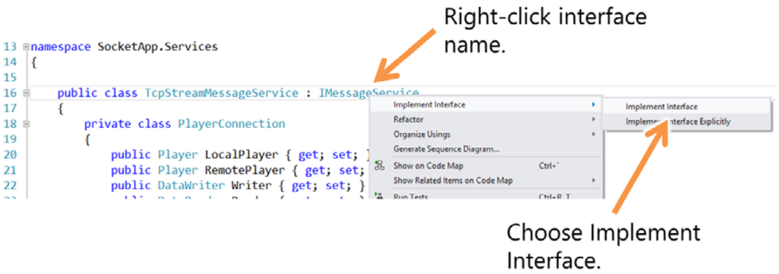
Figure 20.3 Set the build action to None on the `UdpMessageService` class because you'll only update the `TcpMessageService` class.

from the project. Either way, this will keep the class from being included in the project's build. Figure 20.3 shows how to do this from the Solution Explorer.

20.3.2 Sending location information

The interface defines how the message services expose functionality to other classes. In this case, you're only using the TCP version, so the `TcpStreamMessageService` needs to implement the new method and event defined in the interface.

An easy way to synchronize the class with the interface is to right-click the interface name in the class declaration, then choose Implement Interface > Implement Interface. Figure 20.4 shows what this looks like.



```
namespace SocketApp.Services
{

    public class TcpStreamMessageService : IMessageService
    {
        private class PlayerConnection
        {
            public Player LocalPlayer { get; set; }
            public Player RemotePlayer { get; set;
            public DataWriter Writer { get; set; }
```

Right-click interface name.

Choose Implement Interface.

Figure 20.4 Implement the interface to pull in the missing event and method declarations.

This will generate the event declaration as well as a stub for the new `SendLocationUpdate` method. The event declaration is fine as is, but the `SendLocationUpdate` method will need to be filled out. Similarly, you'll need to make some changes to the other send methods so they include the location information as part of their streams of data. Following are the changes to the `TcpStreamMessageService` class.

Listing 20.9 Updated message-sending methods in the TCP service class

```
private void WritePlayerLocation(DataWriter writer,
                                 PlayerLocation location)
{
  writer.WriteDouble(location.X);
  writer.WriteDouble(location.Y);
  writer.WriteDouble(location.Angle);
}

private void WritePascalStyleString(string s)
{
  if (s != null && s.Length > 0)
  {
    _connection.Writer.WriteInt32((Int32)s.Length);
    _connection.Writer.WriteString(s);
  }
  else
  {
    _connection.Writer.WriteInt32(0);
  }
}
```

Write location

Write a Pascal-style string

Send player join message

```
private async void SendPlayerJoinMessage()
{
  if (_connection.Writer != null)
  {
    string playerName = "(unknown)";
    if (_connection.LocalPlayer != null)
      playerName = _connection.LocalPlayer.Name;

    _connection.Writer.WriteInt32((Int32)WireMessageType.PlayerJoin);
    WritePascalStyleString(playerName);
    WritePlayerLocation(_connection.Writer,
                        _connection.LocalPlayer.Location);

    await _connection.Writer.StoreAsync();
  }
}
```

Send player leave message (unchanged)

```
private async void SendPlayerLeaveMessage()
{
  if (_connection.Writer != null)
  {
    _connection.Writer.WriteInt32((Int32)WireMessageType.PlayerLeave);

    await _connection.Writer.StoreAsync();
  }
}
```

Send chat message

```
public async void SendChatMessage(ChatMessage message)
{
  if (_connection.Writer != null)
  {
    _connection.Writer.WriteInt32((Int32)WireMessageType.ChatMessage);
    WritePascalStyleString(message.Message);
    WritePlayerLocation(_connection.Writer,
                        _connection.LocalPlayer.Location);

    await _connection.Writer.StoreAsync();
  }
}
```

New send location message

```
public async void SendLocationUpdate()
{
  if (_connection.Writer != null)
  {
    _connection.Writer.WriteInt32((Int32)WireMessageType.PlayerLocation);

    WritePlayerLocation(_connection.Writer,
                        _connection.LocalPlayer.Location);
    await _connection.Writer.StoreAsync();
  }
}
```

All of the send methods are included in this listing. The only new method is `SendLocationUpdate`, but all except the `SendPlayerLeaveMessage` functions include changes either to use the helper methods or to send location information as part of their data stream.

This listing also includes two helper methods. The first `WritePlayerLocation` is a reusable method that standardizes the format of play location data on the wire. It's used by most of the send methods. The second is the `WritePascalStyleString` method. This method standardizes the format of strings on the wire by first writing the length and then writing the message text.

20.3.3 Reading location information

Although you use only one of them for now, several messages contain the `PlayerLocation` data. To make your code more robust, I moved into a separate function the code to parse the player location information. The next listing includes the code to add to the `TcpStreamMessageService` class.

Listing 20.10 New function to read player location data

```
private PlayerLocation ReadPlayerLocation(DataReader reader)
{
  var loc = new PlayerLocation();

  loc.X = reader.ReadDouble();
  loc.Y = reader.ReadDouble();
  loc.Angle = reader.ReadDouble();

  return loc;
}
```

Read properties from stream

This function is used by the `ProcessIncomingMessages` function. The `ProcessIncomingMessages` function is one of the most important functions in the `TcpStreamMessageService` class. In the next listing, it's been updated to handle the location data and the new `PlayerLocation` message.

Listing 20.11 Updated `ProcessIncomingMessages` function

```
private void ProcessIncomingMessages()
{
  var t = Task.Factory.StartNew(async () =>
  {
    _connection.Reader.InputStreamOptions = InputStreamOptions.Partial;

    while (true)
    {
      var count = await _connection.Reader.LoadAsync(MaxMessageSize);

      var messageType = (WireMessageType)_connection.Reader.ReadInt32();

      switch (messageType)
      {
        case WireMessageType.PlayerLeave:
          if (PlayerExited != null)
            PlayerExited(this,
                new PlayerExitedEventArgs(_connection.RemotePlayer));
          break;
        case WireMessageType.PlayerJoin:
          if (PlayerJoined != null)
          {
            var nameLength = _connection.Reader.ReadInt32();
            var name = _connection.Reader.ReadString((uint)nameLength);
            var loc = ReadPlayerLocation(_connection.Reader);

            var remotePlayer = new Player();
            remotePlayer.Name = name;
            remotePlayer.Location = loc;

            _connection.RemotePlayer = remotePlayer;

            PlayerJoined(this, new PlayerJoinedEventArgs(remotePlayer));
          }
          break;

        case WireMessageType.ChatMessage:
          if (ChatMessageReceived != null)
          {
            var msgLength = _connection.Reader.ReadInt32();
            var text = _connection.Reader.ReadString((uint)msgLength);

            var msg = new ChatMessage();
            msg.Message = text;
            msg.Player = _connection.RemotePlayer;

            msg.Player.Location =
                        ReadPlayerLocation(_connection.Reader);

            ChatMessageReceived(this,
               new ChatMessageReceivedEventArgs(msg));
          }
          break;

        case WireMessageType.PlayerLocation:
          var player = _connection.RemotePlayer;
          player.Location = ReadPlayerLocation(_connection.Reader);

          if (PlayerLocationUpdated != null)
            PlayerLocationUpdated(this,
               new PlayerLocationUpdatedEventArgs(player));
          break;
      }
    }
  });
}
```

Read player location

New case

As I mentioned earlier, all messages (except exit) now include the player location. In the app for this book, you're only interested in the one in `PlayerLocation`, although the code does update it for each message. Updating with each message is a common synchronization technique for apps that use sparse updating calls. That is, they update only when major actions happen, such as when a shot is fired, a collision is detected, or something similar.

NOTE Just a reminder that the code in this section isn't suitable for anything with high volumes of data because it assumes one read equals one message. Really robust message-processing code needs to account for incomplete messages and messages that come in more than one to a read.

The event raising is a convenience but isn't actually required when it comes to updating the player's location. Why is that? Because the `Player` instances in the `TcpStreamMessageService` class happen to be the same instances used by the viewmodel and bound to by the UI. Simply updating the properties from within the message service propagates the changes all the way down to the UI.

I'm not super happy about the coupling this imposes on the design, but it reduces the amount of code required in these chapters and is a reasonable design for a small game or sample app.

With the changes in this section, the UI can now respond to the addition and removal of players across the network. It will also respond to location changes sent from the remote machine. This is a significant milestone for this app, because now you're sending more meaningful data than just the chat message. There's real game data going across the wire! Before we wrap up this chapter, you need an easy way to test that everything is working.

20.4 Testing everything

In the next chapter, you'll use keyboard, mouse, touch, and more to move the ships around the screen. For now, you need another way to test. Rather than add buttons on the screen or some other UI, you'll do something a little more automated and interesting.

Going back to `MainViewModel`, you'll test everything by adding a new function: `TestPositionUpdate`. This was stubbed out earlier in this section, but now you'll finally make use of it. You may recall that this function is called once the app receives notification that another player has joined.

The next listing has the test code to add to the `MainViewModel` class.

Listing 20.12 Testing the position update over the network

```
public void TestPositionUpdate()
{
  var t = Task.Factory.StartNew(async () =>
  {
    for (int i = 0; i < 100; i++)
    {
      DispatcherHelper.CheckBeginInvokeOnUI(() =>
        {
          Player.Location.X += 3;
          Player.Location.Y += 3;
          Player.Location.Angle = i * 5;

          _messageService.SendLocationUpdate();
        });
```

Background thread

Property changes on UI thread

Property changes

Send to other machine

```
        await Task.Delay(500);
      }
    });
}
```

Delay 1/2 second

If you run the app now, you'll see that once you've connected to the other machine, both players will begin to slowly move on the screen, rotating and moving at the same time. If that's what you see, congratulations! You're ready to add humans into the game.

Smoothing out the movement animation

Although you can do a lot with it, XAML is not necessarily the best medium for frame-based animation. For that game type, you'll want to consider using C++ and DirectX—you can still use XAML for much of the UI and even use C# combined with C++ for a bit of both.

To get smoother animation in this app, you may want to consider starting Storyboard instances when new location information is received. Set the duration to the location reporting interval and the final values to those sent in the location information.

Of course, this will place your representation of the game behind by the amount of network delay plus the location-reporting interval.

Another approach would be to send vector information rather than final coordinates. For example, you could send location-reporting information that includes the current location plus direction (angle) and velocity. The app could then interpolate to figure out the location smoothly, again using Storyboards.

All of these are more than I'd want to tackle here, because I think you'll get even more mileage by incorporating a good physics library that understands things like acceleration, deceleration, collision, and more. Those libraries often have their own way of representing frame-based data, so you'd want to accommodate that when designing your messages.

20.5 Summary

This chapter helped flesh out the code for the sample game app you're building as you near the end of this book. Although much of the content here was plumbing-related work, you also learned a few new interesting techniques, like how to use an `ItemsControl` to display arbitrary lists of data in a non-tabular format and how to use background threads to interact with UI objects.

You updated the `Player` class and introduced the new `PlayerLocation` class to help build up the model for this app. These two classes are for the major bit of data that you use both as a client and as a server, so getting them right was very important.

By hanging certain properties off the `PlayerLocation`, such as `X`, `Y`, and `Angle`, you were then able to use those in a data template in the `ItemsControl` in the UI. Other properties, like the name and the color of the player's ship, were exposed directly from the `Player` class.

You then updated the `TcpStreamMessageService` to send and receive this new player data so that both ends of the network conversation will stay in sync.

The basic MVVM approach we've taken so far made it easy for you to test the UI at a couple points throughout this work. You were able to test adding a number of players to the screen and later were able to test updating the positions of two real players.

As I mentioned in the chapter introduction, the main purpose for this chapter was to provide all the in-between code necessary to bridge the gap between a simple TCP sockets network implementation and a more complete game demo. In the next chapter, you'll add some interaction through touch, keyboard, mouse and more. It'll really feel like a game once you're able to move things around.

21

Keyboards, mice, touch, accelerometers, and gamepads

This chapter covers

- Keyboard input
- Touch, mouse, and pen input
- Using the accelerometer
- Integrating C++ to use an Xbox 360 controller

Prior to Windows 8, many tablets supported keyboard input only as an afterthought and mouse or pen input as an anomaly. Similarly, most desktop computers weren't designed with touch input in mind. Each device had its preferred means of input and essentially ignored the others. The apps on those platforms also reflected these input choices.

The WinRT and Windows Store side of Windows 8 has been built from the ground up with the understanding that users will have multiple ways of interacting with the system. We didn't want to restrict tablets to touch or desktops to only mouse. Instead, we understood that there's a continuum of devices that span form factors, and that the line between what is a tablet versus a notebook versus a desktop can sometimes be blurry. For example, when I set up the pre-keynote performance at Build 2012, Jordan Rudess (keyboardist for Dream Theater and a

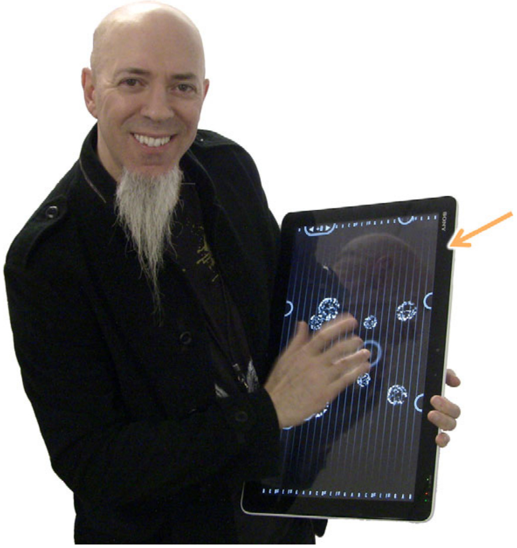

Is this a tablet or a PC? Is the distinction even important?

Figure 21.1 Jordan Rudess playing Wizdom Music Tachyon on a Sony VAIO TAP 20 at Build 2012. The TAP 20 is an all-in one PC with a battery and a Bluetooth keyboard. Is it a tablet? A PC? Both? Should it only support one type of input? (Note: This is not an endorsement of this specific PC.)

performance technology enthusiast) played his tablet app on a 27" touch screen all-in-one computer.[1] This was new territory for him and his developer and something he found really opened up the possibilities of what he'd do with his apps. He also played the same app on Microsoft Surface with Windows RT.

While downstairs in building 92 on campus, Jordan played with a Sony VAIO Tap 20 all-in-one PC onto which I had side-loaded the development version of his performance app Tachyon. This PC (figure 21.1) looks like a giant 20" tablet, is battery powered, but weighs more than most Ultrabooks. Sony intended it as a desktop that you might use to watch movies or serve as a kitchen PC, but Jordan was interested in it for its tablet-like performance qualities. For all you Harry Potter™ fans, I jokingly call it Hagrid's tablet.

By making every device a PC, we've come down firmly on the side of choice of input. In fact, every app developed for the Windows Store must support keyboard input, touch input, and mouse input.

In our app so far, we have a ship, and we have a means for transmitting its location across the network, but we don't yet have any way for a user to modify its location and angle. It's like we've built a car but left the steering wheel off until the end.

In this chapter we'll build the entire input mechanism for the app. First, we'll generalize the input into a single interface. We'll also build out the code in the viewmodel that polls this interface. From there, we'll build our first input service: keyboard input. As part of that, we'll look at local versus global key handlers and the translation from virtual keys from app commands.

---

[1] You can see a video of his first impressions on my YouTube channel, www.youtube.com/user/Psychlist1972.

After the initial keyboard service, we'll look at something that's arguably the most important input mechanism in Windows 8: pointers such as mice, touch, and stylus. From there, we'll move into another really cool type of input: the accelerometer sensor. If you have a tablet or other device that supports tilt and rotation, this section will finally give you an excuse to play with that new toy.

We'll wrap up this chapter with something really outside of the box: using an Xbox 360 controller for Windows. This will involve integrating C++ code and XInput from DirectX but will result in a truly game-like control experience. That was one of my favorite things to do when working on this chapter. I can't tell you how gratifying it was to set my Surface in front of me, plug in an Xbox controller, and move something around the screen in an app I wrote.

21.1 Making input generic

Most games support multiple means of input. This was true going all the way back to the old Commodore 64 games I used to play; those would almost always support both keyboard and joystick input, because not everyone had access to a joystick. Even today, games support multiple platforms, each with its own common input devices. Some have touch screens; others have game pads with different layouts and numbers of buttons; some, like Kinect, even use gesture control. Of course, we also still have the good-old keyboard on laptops, desktops, and sometimes on tablets and phones.

When developing a game, most developers don't special-case each and every input device. Instead, they abstract the input devices away from the commands they represent. In that way, the app code doesn't need to deal with the user pressing the A button on their controller or the spacebar on their keyboard—the app only knows the user invoked the Jump command. We'll do the same here, so that we can support keyboard, pointers, sensors, and more.

In this section, you'll put together the plumbing of the input "stack" for your game. You won't deal with specific input from any one device but will instead create the generic handling that will enable you to acquire input from any device you decide to incorporate. A core part of this is the `IInputService` interface, so let's start there.

21.1.1 The IInputService interface

The `IInputService` defines the shape of an input service. The intent of the service is to be used in a polling loop, not as a source of events, so it exposes properties for each of the possible states of the input device. The expected use of this interface is shown in figure 21.2.

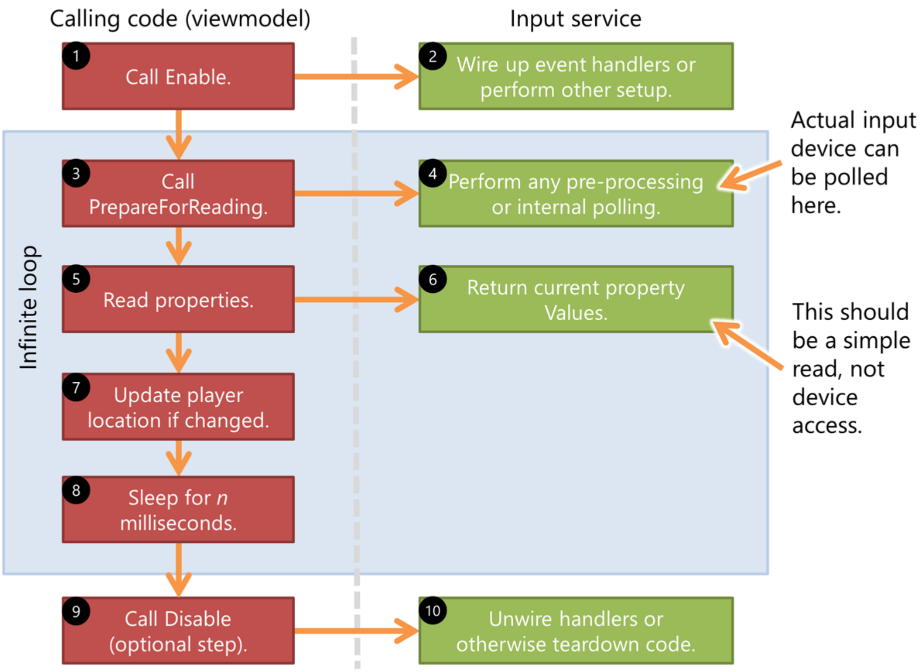

Calling code (viewmodel)
Input service
1 Call Enable.
2 Wire up event handlers or perform other setup.
Infinite loop
3 Call PrepareForReading.
4 Perform any pre-processing or internal polling.
Actual input device can be polled here.
5 Read properties.
6 Return current property Values.
This should be a simple read, not device access.
7 Update player location if changed.
8 Sleep for *n* milliseconds.
9 Call Disable (optional step).
10 Unwire handlers or otherwise teardown code.

Figure 21.2 The `IInputService` flow when used from the viewmodel in our app

The code that defines the interface is shown in the first listing.

Listing 21.1 The `IInputService` interface

```
using SocketApp.Model;
using System;
using Windows.UI.Xaml.Controls;

namespace SocketApp.Services
{
  public interface IInputService
  {
    void Enable(Control hostControl);
    void Disable();

    void PrepareForReading(PlayerLocation currentLocation);

    bool IsMovingForward { get; set; }
    bool IsTurningClockwise { get; set; }
    bool IsTurningCounterClockwise { get; set; }
  }
}
```

Setup

Teardown

Poll or prepare

Properties

The app will use the interface to find out what input devices are in what states and use that information to reposition the ship. If you add more to the game, such as the ability to shoot or move backward, for example, you'll want to add properties like `IsFireButtonPressed` or `IsMovingBackward` and then implement them for each input device.

21.1.2 A little math help

The input devices tell you whether to move the ship forward or turn it (counter)clockwise. In order to realize that movement, you need to write code to calculate the new position of the ship, given its current angle and an arbitrary distance. If the ship only supported horizontal and vertical movement, the math to figure out the next position would be very simple. But because you can support moving in any arbitrary direction, you need to do a little trigonometry.

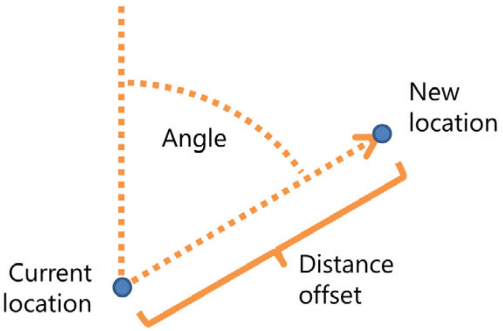


Figure 21.3 Calculating the new location given the angle and a distance

Figure 21.3 shows what you're given (the current location, angle, and distance offset) and what you need to calculate (the new location).

I don't know about you, but it has been many years since I learned SOH-CAH-TOA in high school trigonometry class, so I had to rely on a little WIK-IPE-DIA (and even a question I posted to math.stackexchange.com[2]) to remind me how to use these basic trig functions.[3] The next listing has the `MathService` class with a single `CalculateNewLocation` method that handles everything for you.

Listing 21.2 The `MathService` class

```
using SocketApp.Model;
using System;
using Windows.Foundation;

namespace SocketApp.Services
{
  static class MathService
  {
    public static PlayerLocation CalculateNewLocation(
            PlayerLocation currentLocation,
            double distance)
    {
      var loc = new PlayerLocation();

      loc.Angle = currentLocation.Angle % 360;
```

---

[2] "Calculating an angle adjacent to hypotenuse given two points," Mathematics Stack Exchange, Dec. 16, 2012, http://bit.ly/PeteForgotMath.

[3] The day Wikipedia blacked out its site to protest SOPA in the United States, the perceived IQ of people answering questions on the internet dropped by a good 60 points.

Convert to radians

Calc new position

```
      var theta = loc.Angle * Math.PI / 180.0;

      loc.X = currentLocation.X + distance * Math.Sin(theta);
      loc.Y = currentLocation.Y - distance * Math.Cos(theta);

      return loc;
    }
  }
}
```

The code in this method uses a little geometry and trigonometry to calculate the new location. This method is sufficient for what you'll need to support keyboard input. You'll expand on this method and this service later when we get into touch and the calculation requirements become a bit more complex.

21.1.3 Wiring up the viewmodel

Because you've defined a common interface, you can write the viewmodel code without yet having an actual input implementation.

If you look back at figure 21.2, you'll see that there's a polling loop that will call the input devices. This polling loop exists in the viewmodel. You don't want a tight loop running on the UI thread, so this loop runs on a background thread, with the updates invoked on the UI thread using the `DispatcherHelper`. This invocation is a common technique you've used in several other places in this app.

To make the code easier to understand, I've separated out the single iteration from the overall background loop and implemented it in a separate function. The following listing shows the code for a single iteration. Add this to the `MainViewModel`.

Listing 21.3 The `IInputService` in use in `MainViewModel`

Angle increment

Move increment

```
public List<IInputService> InputServices = new List<IInputService>();
private const double AngleIncrement = 3.0;
private const double MovementIncrement = 5.0;

private void PollAndHandleInputServices()
{
  bool locationChanged = false;
```

Invoke on UI

```
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
    {
      foreach (IInputService input in InputServices)
      {
```

Prepare for polling

```
        input.PrepareForReading(Player.Location);
```

Forward

Use math service

```
        if (input.IsMovingForward)
        {
          Player.Location = MathService.CalculateNewLocation(
            Player.Location, MovementIncrement);
          locationChanged = true;
        }
```

Counterclockwise rotation

```
        if (input.IsTurningCounterClockwise)
        {
          Player.Location.Angle -= AngleIncrement;
          locationChanged = true;
        }
                                                          Clockwise
        if (input.IsTurningClockwise)                     rotation
        {
          Player.Location.Angle += AngleIncrement;
          locationChanged = true;
        }
      }

      if (locationChanged)
      {                                                    Send network
        _messageService.SendLocationUpdate();              update
      }
    });
}
```

The single iteration checks each input device in the collection named `InputServices`. You'll load up that collection later when you have concrete input device implementations.

Once the polling has completed, the function checks to see if the location has changed (via the `locationChanged` flag). If it has changed, the new location information is sent across the wire to update the remote machine.

As I mentioned, this all happens in a loop. That background loop code, also in `MainViewModel`, is shown here.

Listing 21.4 The viewmodel background loop to poll input

```
private const int InputPollingDelayMilliseconds = 100;

private void StartBackgroundInputPolling()                 Spin up
{                                                          background
  var t = Task.Factory.StartNew(async () =>                thread
    {
      while (true)
      {
        PollAndHandleInputServices();                      Poll

        await Task.Delay(InputPollingDelayMilliseconds);   Sleep
      }
    });
}
```

Once started, the background input polling loops for the full lifetime of the viewmodel, which in this case is the lifetime of the app. The `Task.Delay` call as used here is the modern equivalent of the commonly used .NET `Thread.Sleep` function. Note that `Task.Delay` is primarily intended to be used to delay the start of a task. To use it as a sleep function, you must await it and provide no additional actions.

There are a number of places where this background loop could be started. I decided to make it so that the UI didn't respond to any movement input until there were two players on the field. For this reason, the task spin-up happens in the `OnServicePlayerJoined` event handler—the handler that's called when a second player joins the game. The next listing shows the `MainViewModel` modification to support this.

Listing 21.5 Starting background input polling when new player joins

```
void OnServicePlayerJoined(object sender, PlayerJoinedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
    {
      var p = e.Player;
      p.Color = GetNextPlayerColor();

      Players.Add(e.Player);

      ConnectionStatus = "Just joined: " +
            e.Player.Name;
    });

  //TestPositionUpdate();
  StartBackgroundInputPolling();
}
```

Old test code, commented out

New polling task spin-up

Now, when a second player joins, the app will start polling for device input, updating the local player's position and also sending those changes across the wire. As written, this code doesn't scale beyond two players (the polling happens too often, for one thing), but this is sufficient for your uses.

By abstracting away the details and dealing with just an app-optimized interface, you've made it easy to support any type of input device without changing the core app code. The viewmodel deals only with the interface and its app-specific input state properties. I know that interface-based development can be taken to painful extremes—and often is—but this is one place where it makes complete sense and benefits the architecture of the app.

All you have in place at this point is plumbing. For this to work, you need a concrete implementation of `IInputService`. We'll start with the easiest: keyboard input.

21.2 Keyboard input

The keyboard is one of the oldest human input devices for computers. About the only things that predate it are punch cards and physical bit switches. Even in a world increasingly dominated by touch and gesture control, the keyboard remains one of the most important types of input.

In our app, you're going to use the keyboard to control movement and rotation, so only a few keys will be needed, as shown in figure 21.4.

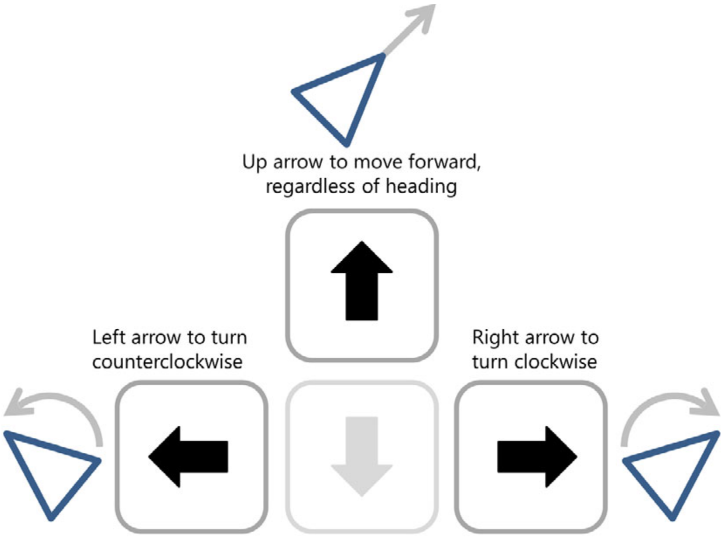

Up arrow to move forward, regardless of heading
Left arrow to turn counterclockwise
Right arrow to turn clockwise

Figure 21.4 The cursor keys are the means of keyboard control for this app. Only three of the four keys are used, because we don't support moving backward. The up cursor key moves forward, the left cursor key turns counterclockwise, and the right cursor key turns clockwise.

You'll start by creating the `KeyboardInputService`—a class that implements `IInputService`. Next, you'll write the key translation code, turning virtual keys into the information required by the app. Finally, you'll work to hook up keyboard events both on a specific control and globally across the app.

21.2.1 The KeyboardInputService

The `KeyboardInputService` is the first test of our new `IInputService` interface. It needs to take key input from the page and translate that into values that indicate the turn direction (if any) and whether or not the ship should be moving forward.

The following listing has the shell of the `KeyboardInputService`. Create this class in the same Services folder as the `IInputService` interface.

Listing 21.6 Shell of the `KeyboardInputService`

```
using SocketApp.Model;
using System;
using System.Threading.Tasks;
using Windows.System;
using Windows.UI.Core;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Input;

namespace SocketApp.Services
{
  public class KeyboardInputService : IInputService
  {
    private Control _host;
    public bool IsTurningCounterClockwise { get; set; }
    public bool IsTurningClockwise { get; set; }
    public bool IsMovingForward { get; set; }

    private bool _isGlobalKeyHook = false;

    private PlayerLocation _currentLocation;
    public void PrepareForReading(PlayerLocation currentLocation)
    {
      _currentLocation = currentLocation;
    }

    public void Enable(Control hostControl)
    {
      if (_host != null || _isGlobalKeyHook)
        Disable();

      if (hostControl != null)
      {
        _host = hostControl;

        _host.KeyDown += OnControlKeyDown;
        _host.KeyUp += OnControlKeyUp;
      }
      else
      {
        _isGlobalKeyHook = true;
        Window.Current.CoreWindow.KeyDown += OnCoreWindowKeyDown;
        Window.Current.CoreWindow.KeyUp += OnCoreWindowKeyUp;
      }
    }

    void OnCoreWindowKeyDown(CoreWindow sender, KeyEventArgs args) { }
    void OnCoreWindowKeyUp(CoreWindow sender, KeyEventArgs args) { }
    void OnControlKeyDown(object sender, KeyRoutedEventArgs e) { }
    void OnControlKeyUp(object sender, KeyRoutedEventArgs e) { }

    public void Disable()
    {
      if (_host != null)
      {
        _host.KeyDown -= OnControlKeyDown;
        _host.KeyUp -= OnControlKeyUp;
        _host = null;
      }
      else if (_isGlobalKeyHook)
      {
        Window.Current.CoreWindow.KeyDown -= OnCoreWindowKeyDown;
        Window.Current.CoreWindow.KeyUp -= OnCoreWindowKeyUp;
        _isGlobalKeyHook = false;
      }
    }
  }
}
```

Hook local key events

Hook global key events

Unwire event handlers

Note the event wire-up. For our app, you're going to use the core window event wire-up so it doesn't matter which control has focus; the core window is global to the app.

But the `Control` class itself also supports keyboard events, so I've included support for that as well.

> **NOTE** Including global keyboard input as well as the chat text entry field on the same screen is a recipe for problems. If you need global handling in your own app, you may consider switching it on and off depending on whether the text entry field has focus, especially if you're using keys like the spacebar or letter keys.

The event handlers are empty at the moment; you'll complete those shortly. First, you need to look at how to handle the keys that are given to you from those handlers. Both events provide the key information in the form of a virtual key.

21.2.2 Virtual keys

A *virtual key* is a representation of a keyboard key that may exist on a physical keyboard or an onscreen keyboard. It's virtual because the key itself is mapped. Not only is the key one that may not exist on a physical keyboard (like numeric keypad keys on most notebooks), but it may be a combination of keys. One such example is `VirtualKeys.Shift`, which can represent either `VirtualKeys.LeftShift` or `VirtualKeys.RightShift` for apps that don't need to differentiate.

For our app, you need to handle only the three cursor keys. The virtual keys for these are `Left`, `Right`, and `Up`. You'll use constants to represent these keys so that you can easily change the key assignment. The next listing shows the key mapping.

Listing 21.7 Key processing

```
private const VirtualKey KeyTurnCounterClockwise = VirtualKey.Left;
private const VirtualKey KeyTurnClockwise = VirtualKey.Right;
private const VirtualKey KeyMoveForward = VirtualKey.Up;

private void HandleVirtualKeyDown(VirtualKey key)
{
  switch (key)
  {
    case KeyTurnCounterClockwise:
      if (!IsTurningCounterClockwise)
        IsTurningCounterClockwise = true;
      break;

    case KeyTurnClockwise:
      if (!IsTurningClockwise)
        IsTurningClockwise = true;
      break;

    case KeyMoveForward:
      if (!IsMovingForward)
        IsMovingForward = true;
      break;
  }
}
```

Recognized keys

Handle key down

```
private void HandleVirtualKeyUp(VirtualKey key)
{
  switch (key)
  {
    case KeyTurnCounterClockwise:
      if (IsTurningCounterClockwise)
        IsTurningCounterClockwise = false;
      break;

    case KeyTurnClockwise:
      if (IsTurningClockwise)
        IsTurningClockwise = false;
      break;

    case KeyMoveForward:
      if (IsMovingForward)
        IsMovingForward = false;
      break;
  }
}
```

Handle key up

This code checks the provided key value when the key is pressed (`HandleVirtualKeyDown`) and when the key is released (`HandleVirtualKeyUp`). When the key is down, the related movement property is set to true. When the key is up, the property is set to false.

So now you know how the keys will be mapped, but you haven't handled the implementation of the keyboard event handlers. In the first listing in this section, you wired up the event handlers for either the control provided to the constructor or the global key hook. The event signatures differ because one is a XAML app model event handler (the control event) and the other is a core window event handler. The core window event handler and the control event handler both supply virtual keys although the property name is different.

The following listing shows both the control and the core window event handler implementations.

Listing 21.8 The keyboard event handlers in the `KeyboardInputService` class

```
void OnCoreWindowKeyDown(CoreWindow sender, KeyEventArgs args)
{
  HandleVirtualKeyDown(args.VirtualKey);
}
void OnCoreWindowKeyUp(CoreWindow sender, KeyEventArgs args)
{
  HandleVirtualKeyUp(args.VirtualKey);
}
```

Global key events

```
void OnControlKeyDown(object sender, KeyRoutedEventArgs e)
{
  HandleVirtualKeyDown(e.Key);
}

void OnControlKeyUp(object sender, KeyRoutedEventArgs e)
{
  HandleVirtualKeyUp(e.Key);
}
```

Local key events

The event handlers for the local and global key hooks (remember, only one of them—either local or global—is active at any given time) call the previously defined `HandleVirtualKeyDown` and `HandleVirtualKeyUp` methods to do the heavy lifting.

Invoking the onscreen keyboard

Text input controls automatically invoke the onscreen keyboard if no other keyboard is available when the control is in focus.

Because you use keyboard input primarily as an alternative means of input versus touch, it makes no sense to use the onscreen keyboard for this app, but it may make sense in your own apps. For example, I'm working on yet another Commodore 64 emulator, and I need to display the onscreen keyboard if no other keyboard is available. I can't use a `TextBox` to cleanly handle this, so I may need to display the onscreen keyboard (although a dedicated Commodore 64-style keyboard would be a better choice).

Code can't arbitrarily display the onscreen keyboard. Instead, the keyboard is displayed because the currently focused control has an automation peer that supports `ITextProvider` (and `IValueProvider`). The built-in text controls have automation peers that implement these required interfaces.

An example of this may be seen in `CustomControl2` in the Touch Keyboard Windows SDK sample on MSDN at http://bit.ly/TextAutomationPeerExample.

21.2.3 *Adding from the code-behind*

Finally, you need to create an instance of the `KeyboardInputService` and load it from the code-behind of the MainPage.xaml.cs file. This isn't the only place where this could be done, but I felt this was a good location to handle input wire-up.

The next listing shows the changes to the `MainPage` code-behind. Be sure to use Alt-F10 or right-click and resolve (automatically add the correct `using` statements) for the `MainViewModel` and `KeyboardInputService` types.

Listing 21.9 MainPage.xaml.cs changes to load the `KeyboardInputService`

```
public MainPage()
{
  this.InitializeComponent();

  Loaded += MainPage_Loaded;
}

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
  var vm = DataContext as MainViewModel;
  vm.InputServices.Clear();

  var keyInput = new KeyboardInputService();
  keyInput.Enable(null);

  vm.InputServices.Add(keyInput);
}
```

Wire up page loaded handler

Loaded handler

Get viewmodel instance

Create keyboard input service

Enable global keyboard hook

Add to input services

Play with the app a little and see how the keyboard responds. If you want the movement to happen faster, you can adjust either the movement increments (make them larger) or the polling interval (make it shorter). Keep in mind that the polling interval controls how often network updates are sent, so that can overwhelm the network stack if you have too tight of a loop, especially on Windows RT-based devices.

Keyboard is the most basic of input types. For many games, however, we'd prefer to use a mouse or touch.

21.3 Pointer input: mouse, touch, and pen

When you think of a tablet's human interface mechanisms, probably the first one that comes to mind is touch. Windows 8 offers a unified touch, mouse, and pen API known as the Pointer API. You can expand on this with touch-specific gestures (we're not going to do that here), or you can treat all of these pointing devices as identical, as we do in this section.

Although touch, pen, and mouse all share the same pointer API, there are some important differences. For example, touch gives you discrete touch points and doesn't otherwise track movement across the screen unless you're using higher-order gestures. The mouse, on the other hand, can track movement anywhere within the app. Touch also doesn't natively have a right-click feature. Because of these differences, features such as displaying something on "hover" make sense for the mouse but not at all for touch or pen input. For that reason, to fully support touch input, you need to ensure that you're not relying on hover states or right-click for critical functionality, although there are equivalents for the latter (hold, touch and tap, and so on).

In this section you'll build out the pointer interface for the app. The interface will allow the user to touch the screen or click the mouse to both turn the ship and move it toward that point. This will require more complex math than what you've used previously, so we'll start there.

21.3.1 Some more math

The most difficult part of this input service will be the calculations involved, because the APIs themselves are quite simple to use. Figure 21.5 shows what you'll need to calculate.

If you click the mouse or touch the pointer in a location counterclockwise from the current heading, you'll need to rotate left. You do the opposite if you touch at a point clockwise from the current heading. You'll also need to include movement as part of that process.

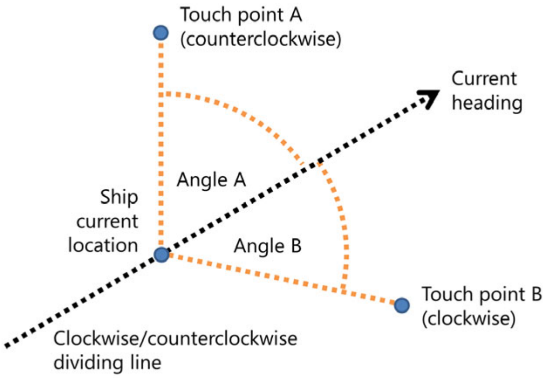
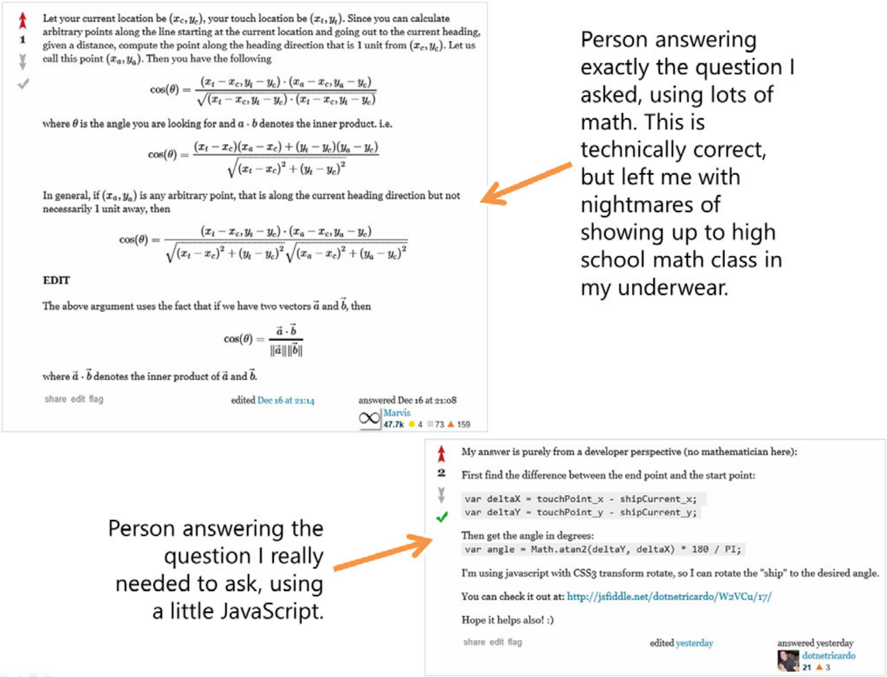


Figure 21.5 An imaginary dividing line continues through the ship both in front and in back. Touch points clockwise from the front of this line produce positive angles. Touch points counterclockwise from the front of this line produce negative angles.


1
2
share edit flag
share edit flag

Let your current location be (x_c, y_c), your touch location be (x_t, y_t). Since you can calculate arbitrary points along the line starting at the current location and going out to the current heading, given a distance, compute the point along the heading direction that is 1 unit from (x_c, y_c). Let us call this point (x_a, y_a). Then you have the following

$$\cos(\theta) = \frac{(x_t - x_c, y_t - y_c) \cdot (x_a - x_c, y_a - y_c)}{\sqrt{(x_t - x_c, y_t - y_c) \cdot (x_t - x_c, y_t - y_c)}}$$

where θ is the angle you are looking for and $a \cdot b$ denotes the inner product. i.e.

$$\cos(\theta) = \frac{(x_t - x_c)(x_a - x_c) + (y_t - y_c)(y_a - y_c)}{\sqrt{(x_t - x_c)^2 + (y_t - y_c)^2}}$$

In general, if (x_a, y_a) is any arbitrary point, that is along the current heading direction but not necessarily 1 unit away, then

$$\cos(\theta) = \frac{(x_t - x_c, y_t - y_c) \cdot (x_a - x_c, y_a - y_c)}{\sqrt{(x_t - x_c)^2 + (y_t - y_c)^2}\sqrt{(x_a - x_c)^2 + (y_a - y_c)^2}}$$

EDIT

The above argument uses the fact that if we have two vectors $\vec{a}$ and $\vec{b}$, then

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

where $\vec{a} \cdot \vec{b}$ denotes the inner product of $\vec{a}$ and $\vec{b}$.

edited Dec 16 at 21:14

answered Dec 16 at 21:08
Marvis
47.7k ● 4 ■ 73 ▲ 159

Person answering exactly the question I asked, using lots of math. This is technically correct, but left me with nightmares of showing up to high school math class in my underwear.

My answer is purely from a developer perspective (no mathematician here):

First find the difference between the end point and the start point:

```
var deltaX = touchPoint_x - shipCurrent_x;
var deltaY = touchPoint_y - shipCurrent_y;
```

Then get the angle in degrees:

```
var angle = Math.atan2(deltaY, deltaX) * 180 / PI;
```

I'm using javascript with CSS3 transform rotate, so I can rotate the "ship" to the desired angle.

You can check it out at: http://jsfiddle.net/dotnetricardo/W2VCu/17/

Hope it helps also! :)

edited yesterday

answered yesterday
dotnetricardo
21 ▲ 3

Person answering the question I really needed to ask, using a little JavaScript.

Figure 21.6 Often, knowing the right question to ask is the trick to getting the answer you need. Either that or finding a helpful community member like @dotnetricardo who can read your mind instead of your question.

USING ATAN2

I struggled with a number of ways of doing this, but in the end a community member came up with a dead-simple solution. Here's a tip: Never ask what's ultimately a programming question on a math board. Figure 21.6 shows why.

Given the information from the math.stackexchange.com question, I did some digging into the `System.Math.Atan2` function. I came up with a set of sample points and then ran the calculations. Figure 21.7 shows the sample data I created.

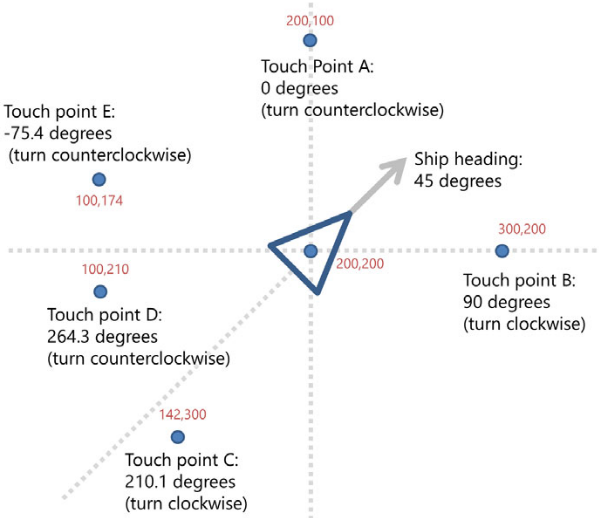

200,100
Touch Point A:
0 degrees
(turn counterclockwise)
Touch point E:
-75.4 degrees
(turn counterclockwise)
100,174
Ship heading:
45 degrees
300,200
100,210
200,200
Touch point B:
90 degrees
(turn clockwise)
Touch point D:
264.3 degrees
(turn counterclockwise)
142,300
Touch point C:
210.1 degrees
(turn clockwise)

Figure 21.7 Test data for trying out the `Atan2` function. The results aren't perfect because of the way the angles are measured (from the x-axis rather than the y-axis), but they can be massaged to get what you want.

And here's how I tested it out. I called `System.Math.Atan2` directly from the command window in Visual Studio 2012 like this:

```
>? System.Math.Atan2(150-200, 250-200) * 180 / System.Math.PI + 90
45.0
>? System.Math.Atan2(100-200, 200-200) * 180 / System.Math.PI + 90
0.0
>? System.Math.Atan2(200-200, 300-200) * 180 / System.Math.PI + 90
90.0
>? System.Math.Atan2(300-200, 142-200) * 180 / System.Math.PI + 90
210.11373315098245
>? System.Math.Atan2(210-200, 100-200) * 180 / System.Math.PI + 90
264.28940686250036
>? System.Math.Atan2(174-200, 100-200) * 180 / System.Math.PI + 90
-75.425783801961273
```

The first entry is the ship heading, the subsequent entries are for touch points A, B, C, D, and E. The form of the call is this:

```
Math.Atan2(touchY-shipY, touchX-shipX) * 180 / Math.PI + 90
```

The reason for the + 90 is because `Atan2` calculates the angle based on the angle from the x-axis with east being zero. For our transform, I need the angle from the y-axis with north being zero. What I would think of as 80 degrees from the vertical is returned by `Atan2` as 10 degrees above the horizontal.

Adding 90 degrees skews the results a little (notice the negative angle for the upper-left quadrant), but you can work with that. Table 21.1 shows the results when you subtract the ship's angle from the touch point angle returned from `Atan2`.

Table 21.1 The results of subtracting the ship's 45-degree heading angle from the `Atan2` touch angle result

Touch point angle	Difference
A (0)	-45
B (90)	+45
C (210.1)	+165.1
D (264.3)	+219.3
E (-75.4)	-120.4

If you assume that negative means counterclockwise and positive means clockwise, there's one result that doesn't work out. Specifically, touch point D should be a counterclockwise move. Similarly, if angle E were normalized to be 360 minus 75.4, or 284.6, rather than a negative value, you'd also get an incorrect result.

NORMALIZING THE RESULTS

The solution is to normalize the angles so that negative values are modified to their positive 360-degree version. For example, a -45 degree angle turns into a positive 315 degree angle. Then, anything over 180 degrees becomes a counterclockwise move, and anything less than 180 degrees becomes a clockwise move (after taking into account the threshold). Table 21.2 shows the final calculations for the test data.

Table 21.2 The final shaping of the results

Touch point angle	Difference	Normalized difference	> 180?	Direction
A (0)	-45	315	Yes	-1
B (90)	+45	45	No	+1
C (210.1)	+165.1	165.1	No	+1
D (264.3)	+219.3	219.3	Yes	-1
E (-75.4)	-120.4	239.6	Yes	-1

Now that you understand the algorithm to make the choice between clockwise and counterclockwise rotation, you can implement it in C#.

TRANSLATING INTO CODE

The hardest part of the algorithm was simply figuring it out. The code itself, once you know the rules, is really simple. The next listing shows the implementation using a new method added to the `MathService` class.

Listing 21.10 New `MathService` method for calculating rotation direction

```
public static int GetAngleDirection(
          PlayerLocation shipLocation,
          Point touchPoint,
          double threshold)
{
  var deltaX = touchPoint.X - shipLocation.X;
  var deltaY = touchPoint.Y - shipLocation.Y;

  var shipAngle = shipLocation.Angle % 360;

  if (shipAngle < 0)
    shipAngle = 360 + shipAngle;

  var touchAngle = Math.Atan2(deltaY, deltaX) * 180 / Math.PI + 90;

  var difference = touchAngle - shipAngle;

  if (Math.Abs(difference) > threshold)
  {
    if (difference < 0)
      difference = 360 + difference;

    if (difference < 180)
      return 1;
    else
```

Normalize ship angle

Atan2 calculation

Difference

Check threshold

Normalize Atan2 result

Clockwise

```
      return -1;                              Counterclockwise
    }
    else
    {
      return 0;                               Straight
    }
}
```

This method implements the normalization and calculations described so far. Before you use this method from the pointer service, you need to make one minor change elsewhere in the project.

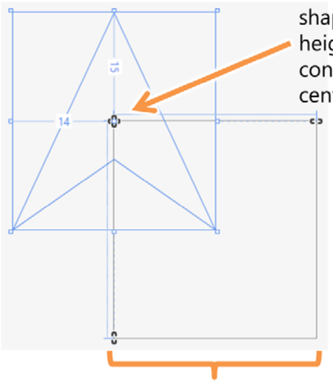
21.3.2 A minor modification to the ship user control

The location of the ship has, so far, been measured from the top left of the user control. That's not perfect, but it worked for everything up until now. Now that you're calculating angles based on the ship's location, that location really needs to be in the center of the ship.

Luckily, this is a very easy fix. Simply crack open the `Ship` user control and, to the `Path` statement, add the following:

```
Margin="-14,-15"
```

Adding this `Margin` property offsets the ship by 50% of its width (28) and height (30), as shown in the Visual Studio designer screen shot in figure 21.8.



Offset ship by half of shape width and height to put user control origin at center point of shape.

Usercontrol root grid bounds

Figure 21.8 Using negative margins, I offset the ship shape to center it at the origin of the user control. This will improve the accuracy of the movement based on the angle calculations.

With that minor change in place, all ship movement will now be relative to the center of the ship. That's exactly what you want.

21.3.3 The PointerInputService class

The `PointerInputService` class follows the same general pattern as the keyboard input service class. It implements the same interface and also uses events to track activity. Event wire-up happens in the `Enable` method and tear-down in the `Disable` method.

The next listing has the full class. Create this class in the Services folder alongside the interface and the `KeyboardInputService` class.

Listing 21.11 `PointerInputService`

```
using SocketApp.Model;
using System;
using System.Threading.Tasks;
using Windows.Foundation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Input;

namespace SocketApp.Services
{
  public class PointerInputService : IInputService
  {
    public bool IsTurningCounterClockwise { get; set; }
    public bool IsTurningClockwise { get; set; }
    public bool IsMovingForward { get; set; }

    private PlayerLocation _currentLocation;
    public void PrepareForReading(PlayerLocation currentLocation)
    {
      _currentLocation = currentLocation;
    }

    private Control _host;
    public void Enable(Control hostControl)
    {
      _host = hostControl;

      if (_host != null)
      {
        _host.PointerPressed += OnPointerPressed;
        _host.PointerReleased += OnPointerReleased;
        _host.PointerMoved += OnPointerMoved;
      }
    }

    private bool _isPointerPressed = false;

    void OnPointerPressed(object sender, PointerRoutedEventArgs e)
    {
      _isPointerPressed = true;
      UpdateState(e.GetCurrentPoint(_host).Position);
    }

    void OnPointerReleased(object sender, PointerRoutedEventArgs e)
    {
      _isPointerPressed = false;

      UpdateState(e.GetCurrentPoint(_host).Position);
    }

    void OnPointerMoved(object sender, PointerRoutedEventArgs e)
    {
      if (_isPointerPressed)
        UpdateState(e.GetCurrentPoint(_host).Position);
    }

    private const double TurnThreshold = 10;

    private void UpdateState(Point point)
    {
```

Wire up events

Pointer pressed

Get pointer position

Pointer released

Pointer moved

"Move forward" angle zone

```
            IsTurningClockwise = false;
            IsTurningCounterClockwise = false;
            IsMovingForward = false;

            if (_isPointerPressed)
            {
              var direction = MathService.GetAngleDirection(
                _currentLocation, point, TurnThreshold);

              IsMovingForward = direction == 0;
              IsTurningClockwise = direction > 0;
              IsTurningCounterClockwise = direction < 0;
            }
          }

          public void Disable()
          {
            if (_host != null)
            {
              _host.PointerPressed -= OnPointerPressed;
              _host.PointerReleased -= OnPointerReleased;
              _host.PointerMoved -= OnPointerMoved;
            }
          }
        }
      }
```

Get movement direction

Clean-up

In addition to the wire-up and tear-down code similar to what you did with the keyboard handlers, this code has an `UpdateState` method. This method, which is called by each pointer event, checks the position of the point in relation to the ship, using the `MathService` method you created earlier. Using that, it sets the direction of rotation, or the movement property.

21.3.4 Adding from the code-behind

Before you can create the service from the code-behind, you'll need to provide a control against which all touch points will be compared. In this case, the `ItemsControl` can't be used because the `ItemsControl`'s default template doesn't help you out by providing any sort of hit test area. You could modify the template for this control or do as I've done and create a quick new `UserControl` that is to serve solely as a hit test area and source of events.

In the Controls folder, create a user control named PlayField.xaml. The markup for this control is shown here.

Listing 21.12 PlayField.xaml user control

```
<UserControl
  x:Class="SocketApp.Controls.PlayField"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:SocketApp.Controls"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  IsHitTestVisible="True">

  <Grid Background="#FF000000">

  </Grid>
</UserControl>
```

Make hit testable

Non-transparent color is essential

The user control simply provides a place for you to hang pointer events on. There are two essential parts of this markup:

- *The control must be hit testable.*

  Most are by default, but it doesn't hurt to be explicit.
- *The control must not be completely transparent.*

  Even if you have a tiny bit of color like #01000000 it will work, but you can't have #00 as the alpha component or have `Opacity` set to 0.0.

The next listing shows where this new control fits in on MainPage.xaml. It's in the `Grid` right before the `ItemsControl`.

Listing 21.13 MainPage.xaml update to add the playing field control

```
...
<Grid Grid.Column="1"
      x:Name="PlayArea"
      Margin="10,10,10,0">
  <Viewbox>
    <Grid Width="1024" Height="658">
      <Rectangle StrokeThickness="4"
                 Stroke="{StaticResource AccentBrush}" />

      <controls:PlayField HorizontalAlignment="Stretch"
                          VerticalAlignment="Stretch"
                          Margin="2"
                          x:Name="PlayField" />

      <ItemsControl ItemsSource="{Binding Players}"
                    HorizontalAlignment="Stretch"
                    VerticalAlignment="Stretch"
                    Margin="2">
...
```

PlayField control

With the playing field control in place, you can then create an instance of the `PointerInputService` inside the code-behind. This follows exactly the same pattern as the `KeyboardInputService` but with the control instance passed in rather than null. The following listing has the updated `Loaded` event handler.

Listing 21.14 MainPage.xaml.xs update to add the service

```
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
  var vm = DataContext as MainViewModel;
  vm.InputServices.Clear();
```

```
    var keyInput = new KeyboardInputService();
    keyInput.Enable(null);

    var pointerInput = new PointerInputService();
    pointerInput.Enable(PlayField);

    vm.InputServices.Add(keyInput);
    vm.InputServices.Add(pointerInput);
}
```

Create PointerInputService

Initialize with control

Add to list

Now, try out the app. As long as you're connected to a server, when you touch a point beyond the ship or move your mouse around with the button held down, you should see the ship start moving in that direction. If you touch the screen and hold your finger there, the ship will turn until it reaches that angle and then will start moving forward. Admittedly, the movement is neither smooth nor flawless, but that's more an issue with my layout and algorithms than it is with the input APIs.

> **NOTE** If you're not connected to a server and you try to move the ship, you'll get an exception due to the `_currentLocation` being null. The disconnected case is not in scope for this app.

If you find the ship turns a little too much like a slow ocean liner, increase the angle change amount in `MainViewModel`. If you find the "move forward zone" too large or small, feel free to change that as well.

Touch, pen, and mouse are all together referred to as pointers in WinRT. Each of these pointers can give you an exact coordinate on the screen for a touch (or click, or tap). The Pointer API gives you quite a bit to work with, but the majority of developers will simply use coordinates and touch points for most apps, as I did here. For further investigation, check out the pointer and related events on the `Control` base class in MSDN. You'll find functions for capturing the pointer, as well as a large number of APIs for capturing taps, double-taps, "right" taps, drag and drop, manipulation (gestures), and much more. The MSDN page is here: http://bit.ly/Win8XamlControlClass.

CoreWindow and the pointer

The `CoreWindow` class that you used for keyboard input also has pointer events. It's not as useful for the typical app developer, however, because it captures all pointer input—even that over the app bar and other controls in the UI. In most cases, you'll want to restrict input capture to a single control on the screen.

In some cases, especially full-screen games with no app bar, the `CoreWindow` pointer events can be useful. They can also be useful if you need to create your own app bar–like capabilities and deal with features like light dismiss and other out-of-bounds pointer events.

21.4 Accelerometer input

As someone interested in technology, you likely realize by now that the sensor that supports screen rotation is not a microscopic person with a sensitive inner ear. The sensor is called an accelerometer.

Accelerometers in your phones and portable computers record G forces on at least two, and more often on three, axes. Either x and y or x, y, and z. The sensor doesn't actually report the angle your device is leaning but rather the G forces experienced on a specific axis. Despite the name, accelerometers don't actually measure acceleration but rather the force in a direction. From this, accelerometers in devices that need actual acceleration data (vehicles, rockets, and so on) can calculate acceleration given the force. In the case of devices, you're typically interested in just the base force bit, not calculating any acceleration over time.

How do they work?

Accelerometers use a number of different technologies to calculate the force, but in general, they come down to a tiny semiconductor mass, suspended inside a chip. That mass moves very small distances based on the movement of the device. Those movements cause a change in capacitance, which is reported back to the main processor as a value.

These chips are incredibly small devices, so the distances moved are measured more in microns, and the masses involved are etched directly on the die just as you'd see for a processor.

These accelerometers are tiny (often only a couple mm square in size for the whole package). They also measure relatively small forces, typically up to around 2 g, but sometimes as much as 8 g.

In devices, accelerometers used to be primarily a safety feature, able to park spinning rust hard drives when the device is dropped. These days, that's still a concern, but with solid state drives (SSD), this use is becoming a bit of a novelty. Instead, most accelerometers are used as input devices, as we're doing in this section.

My wife carries a FitBit. That includes a tiny accelerometer used to count steps walked. My children have magic wands that include accelerometers to tell when they're being waved around. My video camera includes automatic image stabilization, also using an accelerometer. Every modern phone, many notebooks, and most tablets all contain accelerometers for various uses.

If you have a tablet like a Surface, you've already seen the accelerometer in action when you changed the orientation of the device. That's such a common use that the functionality for orientation change is baked right in.

In this section, you'll use the accelerometer as a human input device. That is, you'll turn the whole device into a big joystick by implementing it as another `IInputService`. If you don't have a tablet or other device with a built-in accelerometer, this section will be interesting but not practical for you unless you pick up a device. You can get the STMicroelectronics test board, which is a USB HID device recognized by Windows 8. To get it, use your favorite search engine and search for STEVAL-MKI119V1. It's a bit expensive, at around $125 at the time of this writing. If that's more than you want to spend, you could always sneak into your local Microsoft store, sit down with a Surface or other tablet, and when no one is looking, side-load the app from this chapter. That may earn you an extra special trip to mall security, though, so be warned.

21.4.1 Making sense of the input

Just as with keyboard and pointers, the introduction of a new input mechanism starts with figuring out how it will be used. Our input standard supports turning clockwise/counterclockwise and moving forward. You need to map accelerometer input to these mechanisms. I decided that tilting the device left will turn counterclockwise, tilting right will turn clockwise, and tilting forward will move forward. Figure 21.9 shows the relationship between these movements and accelerometer data.

The accelerometer in our case will return negative g-force values for the x-axis when the device is tilted left, positive x when tilted right. When the device is tilted forward, the accelerometer returns positive y g-force values. In typical use, the g-forces are pretty small, with values around 0.25 or so depending on how rough you are on the device.

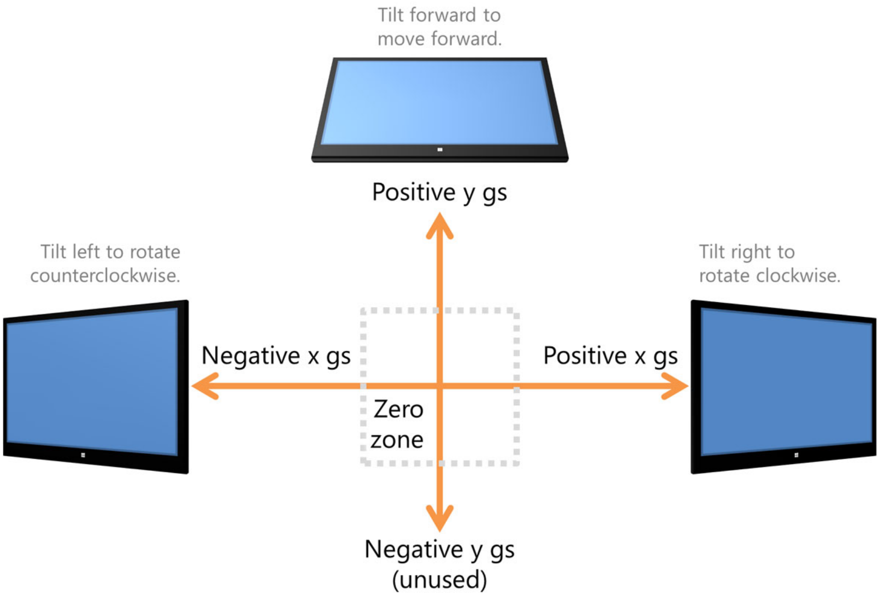

Tilt forward to move forward.
Positive y gs
Tilt left to rotate counterclockwise.
Tilt right to rotate clockwise.
Negative x gs
Positive x gs
Zero zone
Negative y gs (unused)

Figure 21.9 The relationship between ship rotation and movement and the accelerometer. Note the Zero zone (or dead zone) in the middle. When using the accelerometer, you'll want to have a certain movement threshold either from zero or from the last movement to allow for actual human handling of the device without introducing a lot of tiny movements.

21.4.2 Implementing the AccelerometerInputService

Just as with the `KeyboardInputService`, you need a concrete implementation of `IInputService` to enable input from this device. Unlike the keyboard approach, the `AccelerometerInputService` won't use events internally. Instead, it will poll the device inside the `PrepareForReading` method.

The next listing includes the shell of the `AccelerometerInputService`. As before, create this as a class in the Services folder.

Listing 21.15 The `AccelerometerInputService`

```
using SocketApp.Model;
using System;
using Windows.Devices.Sensors;
using Windows.UI.Xaml.Controls;

namespace SocketApp.Services
{
  public class AccelerometerInputService : IInputService
  {
    public bool IsTurningCounterClockwise { get; set; }
    public bool IsTurningClockwise { get; set; }
    public bool IsMovingForward { get; set; }

    private Accelerometer _accelerometer;
    private const int ReportingIntervalMilliseconds = 0;
    public void PrepareForReading(PlayerLocation currentLocation)
    { }

    public void Enable(Control hostControl)
    {
      _accelerometer = Accelerometer.GetDefault();

      if (_accelerometer != null)
        _accelerometer.ReportInterval = ReportingIntervalMilliseconds;
    }

    public void Disable()
    {
      _accelerometer = null;
    }
  }
}
```

Sensor namespace

Accelerometer

Use driver default interval

Get accelerometer

Set interval

Because of the lack of events, the setup and teardown code is far simpler. It creates the accelerometer and sets the interval it should use to check the device. In this case, the interval is set to 0—a value that tells the API to let the driver determine the optimal reporting interval.

Devices like the accelerometer are why I added the `PrepareForReading` method to the interface. In this case, the method does the actual device reading and updates the properties so they have the most recent data when read by the viewmodel code. The following listing has the `PrepareForReading` method with the calls to the accelerometer.

Listing 21.16 Taking the accelerometer reading

```
private const double ReadingThreshold = 0.1;

public void PrepareForReading(PlayerLocation currentLocation)
{
  if (_accelerometer != null)
  {
    var reading = _accelerometer.GetCurrentReading();

    IsTurningClockwise = false;
    IsTurningCounterClockwise = false;
    IsMovingForward = false;

    if (reading.AccelerationX > ReadingThreshold)
      IsTurningClockwise = true;

    else if (reading.AccelerationX < ReadingThreshold * -1)
      IsTurningCounterClockwise = true;

    if (reading.AccelerationY > ReadingThreshold)
      IsMovingForward = true;
  }
}
```

Set zero-area

Get left tilt

Get right tilt

Get forward tilt

The `PrepareForReading` method handles polling the accelerometer for its current values. The `ReadingThreshold` defines the dead zone in the middle of movement, helping to prevent moving when the device is tilted very slightly.

21.4.3 Adding from the code-behind

As before, you'll create an instance of the service in the main page code-behind. The next listing includes the updated code.

Listing 21.17 Updated MainPage.xaml.cs code-behind

```
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
  var vm = DataContext as MainViewModel;
  vm.InputServices.Clear();


  var keyInput = new KeyboardInputService();
  keyInput.Enable(null);

  var accelerometerInput = new AccelerometerInputService();
  accelerometerInput.Enable(null);

  var pointerInput = new PointerInputService();
  pointerInput.Enable(PlayField);

  vm.InputServices.Add(keyInput);
  vm.InputServices.Add(accelerometerInput);
  vm.InputServices.Add(pointerInput);
}
```

Create accelerometer

Add to collection

The code to add the accelerometer looks like what you've done for the other services. There's nothing new and exciting here…until you run the app. Once you run the app, things get really awesome. Try tilting the device left, right, or forward and see what it does to your ship. Also marvel at how the changes are reflected on the other device. If you're not sitting there thinking about possible remote-control scenarios you could create by using a tablet, the accelerometer, and streaming sockets, you likely need a refill on your coffee.

21.4.4 Accelerometer events

The `Accelerometer` class also supports an event-based approach so that you can receive events when the accelerometer data changes. I prefer the polling approach since you're already polling in this app—the approach supported by the majority of input devices, especially game controllers. But for other apps, you may find the event approach a better choice. In those cases, hook up the `ReadingChanged` event.

When going with an event-based approach, you'll want to be very smart about how you set the `ReportInterval`, because events can be costly. You'll need to balance the responsiveness of your app with the workload of dealing with hundreds of events.

Another really interesting event is the `Shaken` event. This event will tell your app when the device has been shaken—a higher-level accelerometer reading than just g-forces on a specific axis. This event could be used to do things like clear a drawing board (think Etch-a-Sketch) or perform some other specific action in your app.

21.4.5 Dealing with screen autorotation

When you work with an accelerometer, the built-in screen autorotation can be a real pain in the button. No one wants the screen to start flipping around when they're playing a pinball game! You could handle this by supporting only a single orientation for your app, but that's not friendly.

Instead, you can lock the orientation of the app programmatically. In the next listing you'll lock it to landscape, but you could read the current orientation and lock to that instead.

Listing 21.18 Locking the screen orientation

```
public void Enable(Control hostControl)
{
  _accelerometer = Accelerometer.GetDefault();

  if (_accelerometer != null)
  {
    _accelerometer.ReportInterval = ReportingIntervalMilliseconds;

    Windows.Graphics.Display
                    .DisplayProperties
                    .AutoRotationPreferences =
           Windows.Graphics.Display
               .DisplayOrientations.Landscape;
  }
}
```

Lock orientation to landscape

You need to be really smart about how you lock the orientation of the app. In fact, I'd recommend making it a user option so they can lock and unlock at will using a button on the app bar or a settings page. In this way, you can let them be in control of the orientation.

Surprisingly, the accelerometer was one of the simplest input devices to use. It's a testament to the API designer's thoughtfulness that something potentially complex was made so easy to use. I'll pass your appreciation on to the team.

21.5 Xbox 360 gamepad input and a little C++

The standard for Windows gamepads is the gamepad introduced with the Xbox 360. DirectX includes built-in support for the Xbox 360 controller for Windows Store apps; see figure 21.10. Because this controller is something typically used just in hard-core games, and those games are almost always written in DirectX; only DirectX has access to the controller.

So how then would one gain access to this controller from a C# and XAML app? This is done through a wrapper class written in C++. This isn't just any old wrapper class, though; this is a class in a WinRT extension library.

WinRT extension libraries can be used by any language that can use WinRT APIs. You could create these libraries in C#, but it's more efficient to create them in C++. In our case, since C++ is the only language with access to XInput (the DirectX controller API), C++ is our only choice.

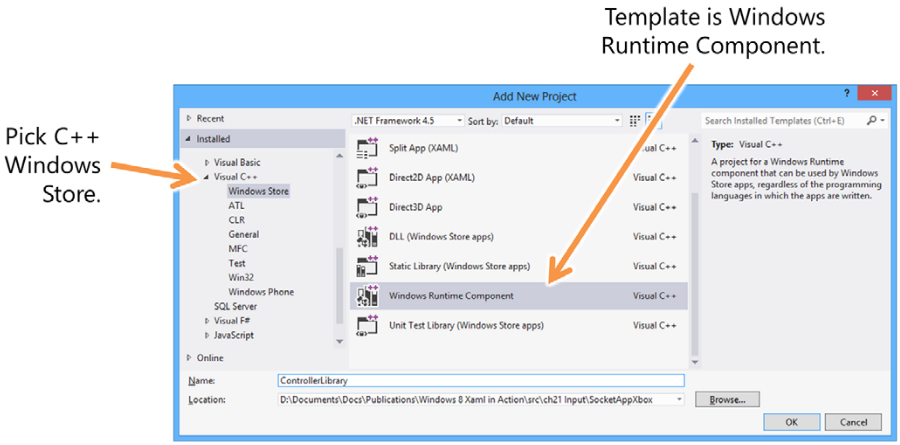
Throughout the rest of this section you'll build out a small C++ WinRT extension library, reference it from the C# app, then create an `IInputService` wrapper class that will expose the capabilities of the controller to the rest of the app using the same approach as all the other input services.

Figure 21.10 The Xbox controller for Windows sitting on my Surface with Windows RT. Yes, the Xbox controller works on Windows RT. You can tell I've had it forever because it's a white one. There are a number of controllers you can use, as long as XInput recognizes them without an additional driver install.

21.5.1 Creating the C++ project

You're going to add a C++ project to the same solution. All versions of Visual Studio 2012 that support building Windows Store apps in C# also enable you to write those apps in C++.

Right-click the solution and select Add New Project. When prompted for a template, select the Windows Runtime Component template, as shown in figure 21.11.


Template is Windows Runtime Component.

Pick C++ Windows Store.

Add New Project
Recent
Installed
Visual Basic
Visual C++
Windows Store
ATL
CLR
General
MFC
Test
Win32
Windows Phone
SQL Server
Visual F#
JavaScript
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Split App (XAML)
Direct2D App (XAML)
Direct3D App
DLL (Windows Store apps)
Static Library (Windows Store apps)
Windows Runtime Component
Unit Test Library (Windows Store apps)
Visual C++
Browse...
OK
Cancel

Type: Visual C++

A project for a Windows Runtime component that can be used by Windows Store apps, regardless of the programming languages in which the apps are written.

Name: ControllerLibrary

Location: D:\Documents\Docs\Publications\Windows 8 Xaml in Action\src\ch21 Input\SocketAppXbox

Figure 21.11 Select the proper template to create a C++ Windows Runtime Component.

You must choose the template under the Visual C++ heading, because we're not creating one in C#. I named my project ControllerLibrary.

Rename Class1.h to Controller.h and Class1.cpp to Controller.cpp. Then, go into the two files and replace `Class1` with `Controller`. The next two listings show what you should have to start with.

Listing 21.19 Controller.h

```
#pragma once

namespace ControllerLibrary
{
    public ref class Controller sealed
    {
    public:
        Controller();
    };
}
```

WinRT extensions must be sealed

Definition of constructor

Listing 21.20 Controller.cpp

```
#include "pch.h"
#include "Controller.h"

using namespace ControllerLibrary;
using namespace Platform;

Controller::Controller()
{
}
```

Include header file

Implementation of constructor

My goal here is not to teach you C++. It's my hope that once you see how you can integrate it into your apps, you'll investigate that on your own. One difference from C#

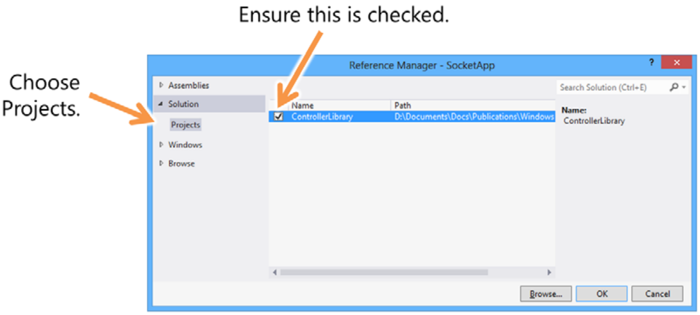

Ensure this is checked.
Choose Projects.
Reference Manager - SocketApp
Assemblies
Solution
Projects
Windows
Browse
Search Solution (Ctrl+E)
Name
Path
☑ ControllerLibrary
D:\Documents\Docs\Publications\Windows
Name:
ControllerLibrary
Browse...
OK
Cancel

Figure 21.12 **Add a project reference to the C++ ControllerLibrary project. Many people simply click the reference and hit OK. Unless you actually check the reference before hitting OK, it won't be added. For some reason, the team took this approach to multiselection.**

that's important to note is the structure of the class files. C# puts both the class definition and its implementation in the same file. C++ breaks it into two files where the definition is contained in the .h (header) file and the implementation is in the .cpp file by convention. A source file can only see types that have been defined in an included header. The pch.h file is a full set of precompiled headers for the project, saving some compile time. This isn't automatically maintained, so you need to maintain it yourself, as shown in the following listing.

Listing 21.21 Updates to pch.h

```
//
// pch.h
// Header for standard system include files.
//

#pragma once

#include <Windows.h>
#include <XInput.h>
```

Standard Windows

XInput for controller access

In general, you only want to put infrequently changing header files in pch.h, because your compile time could go through the roof if you include frequently changing files in there.

Next, in the main SocketApp C# project, right-click the project and select Add Reference, and add a project reference to the ControllerLibrary project, as shown in figure 21.12.

Once you have the project set up and properly referenced from the SocketApp C# app project, you can create the `Controller` class.

21.5.2 *Implementing the Controller class*

The C++ controller header file has two main parts: the `GamepadState` structure and the `Controller` class itself. The `GamepadState` structure, as the name implies, contains the state of the gamepad when polled. It's intended as a friendlier way to get gamepad information without bitmasks and constants as used by the native `XINPUT_STATE` structure.

In addition, the `Controller` class contains a single interesting method: `GetState`. This method polls the gamepad for its current state and then returns it if available.

The entire Controller.h listing is shown here.

Listing 21.22 Controller.h

```
#pragma once

namespace ControllerLibrary
{
  public value struct GamepadState
  {
    bool IsConnected;

    bool IsLeftJoystickPressed;
    short LeftJoystickXValue;
    short LeftJoystickYValue;

    bool IsRightJoystickPressed;
    short RightJoystickXValue;
    short RightJoystickYValue;

    bool IsDPadUpPressed;
    bool IsDPadDownPressed;
    bool IsDPadLeftPressed;
    bool IsDPadRightPressed;

    bool IsAPressed;
    bool IsBPressed;
    bool IsXPressed;
    bool IsYPressed;

    bool IsStartPressed;
    bool IsBackPressed;

    bool IsLeftShoulderButtonPressed;
    bool IsRightShoulderButtonPressed;

    BYTE LeftTriggerValue;
    BYTE RightTriggerValue;

    uint32 ControllerID;
  };

  public ref class Controller sealed
  {
  private:
    XINPUT_STATE _xinputState;

  public:
    Controller();
    GamepadState GetState();
  };
}
```

True if gamepad present

Left thumb stick

Right thumb stick

Dpad

Letter buttons

Middle buttons

Shoulder buttons

Triggers

Controller ID

Raw state data

GetState

The Xbox controller has two analog thumb joysticks plus a D-pad for digital motion. It also has four colored buttons (A, B, X, Y), a back button, a start button, two triggers, and right above the triggers, two shoulder buttons.

Buttons are generally Boolean values. But some buttons, such as the triggers, actually return a value that indicates just how pressed they are. In that way, they're more like potentiometers, like the volume knob on a classic stereo, than straight buttons.

As I previously mentioned, C++ splits the class into two files. Controller.cpp contains the implementation of the `Controller` class defined in the header file. The next listing has the entirety of that implementation.

Listing 21.23 Controller.cpp

```
#include "pch.h"
#include "Controller.h"
#pragma comment(lib, "XInput.lib")

using namespace ControllerLibrary;
using namespace Platform;

Controller::Controller() { }

GamepadState Controller::GetState()
{                                                      Hardcoded to
  const int controllerIndex = 0;                       first gamepad

  GamepadState state;
  state.IsConnected = false;                                       Get raw
                                                                   controller
  DWORD result = XInputGetState(controllerIndex, &_xinputState);   state

  if (result == ERROR_SUCCESS)                              Interpret
  {                                                         data
    state.IsConnected = true;
    state.ControllerID = controllerIndex;

    auto buttons = _xinputState.Gamepad.wButtons;                Get button bits

    state.IsDPadUpPressed = buttons & XINPUT_GAMEPAD_DPAD_UP;
    state.IsDPadDownPressed = buttons & XINPUT_GAMEPAD_DPAD_DOWN;
    state.IsDPadLeftPressed = buttons & XINPUT_GAMEPAD_DPAD_LEFT;
    state.IsDPadRightPressed = buttons & XINPUT_GAMEPAD_DPAD_RIGHT;

    state.IsAPressed = buttons & XINPUT_GAMEPAD_A;
    state.IsBPressed = buttons & XINPUT_GAMEPAD_B;
    state.IsXPressed = buttons & XINPUT_GAMEPAD_X;
    state.IsYPressed = buttons & XINPUT_GAMEPAD_Y;

    state.IsStartPressed = buttons & XINPUT_GAMEPAD_START;
    state.IsBackPressed = buttons & XINPUT_GAMEPAD_BACK;

    state.IsLeftShoulderButtonPressed =
          buttons & XINPUT_GAMEPAD_LEFT_SHOULDER;
```

```
        state.IsRightShoulderButtonPressed =
              buttons & XINPUT_GAMEPAD_RIGHT_SHOULDER;

        state.IsLeftJoystickPressed = buttons & XINPUT_GAMEPAD_LEFT_THUMB;
        state.LeftJoystickXValue = _xinputState.Gamepad.sThumbLX;
        state.LeftJoystickYValue = _xinputState.Gamepad.sThumbLY;

        state.IsRightJoystickPressed = buttons & XINPUT_GAMEPAD_RIGHT_THUMB;
        state.RightJoystickXValue = _xinputState.Gamepad.sThumbRX;
        state.RightJoystickYValue = _xinputState.Gamepad.sThumbRY;

        state.LeftTriggerValue = _xinputState.Gamepad.bLeftTrigger;
        state.RightTriggerValue = _xinputState.Gamepad.bRightTrigger;
      }
      return state;
    }
```

That's the whole class. It really just came down to a single function call and then using built-in constants to interpret the results. The Xbox gamepad is really simple to interface with at this basic level. Next, we'll wrap the class in our usual C# `IInputService` implementing class.

GOING DEEPER WITH THE GAMEPAD CONTROLLER

For more information on `XInput` and the controller programming guide, see http://bit.ly/XInputControllerGuide. I learned how to implement the controller by looking at this guide and also by following the JavaScript Official Windows SDK sample.

There's more to be done to make this an efficient use of the controller (such as limiting how often you poll and dealing with cases where no controller is connected) and even set the rumble motors. You can find out all about that in MSDN.

21.5.3 Creating the IInputService wrapper

Like every other input service, you need to create a class that implements `IInputService`. In the Services folder, create a class file named GamepadInputService.cs and add to it the following code.

Listing 21.24 The `GamepadInputService`

```
using ControllerLibrary;
using System;
using SocketApp.Model;
using Windows.UI.Xaml.Controls;

namespace SocketApp.Services
{
```

The C++ library

```
  class GamepadInputService : IInputService
  {
    private Controller _controller;

    public bool IsTurningCounterClockwise { get; set; }
    public bool IsTurningClockwise { get; set; }
    public bool IsMovingForward { get; set; }

    private PlayerLocation _currentLocation;
    public void PrepareForReading(PlayerLocation currentLocation)
    {
      _currentLocation = currentLocation;

      IsTurningClockwise = false;
      IsTurningCounterClockwise = false;
      IsMovingForward = false;

      var state = _controller.GetState();

      if (state.IsConnected)
      {
        IsTurningClockwise = state.IsDPadRightPressed;
        IsTurningCounterClockwise = state.IsDPadLeftPressed;

        IsMovingForward = state.IsAPressed ||
            state.IsRightShoulderButtonPressed ||
            state.IsLeftShoulderButtonPressed;
      }
    }

    public void Enable(Control hostControl)
    {
      _controller = new Controller();
    }

    public void Disable()
    {
      _controller = null;
    }
  }
}
```

The C++ class

GetState

Turning

Move forward

I decided to use the D-pad for turning and either the A button or either shoulder button for moving forward. Having tried this, I can't tell you how much more instant fun you get from messing with a game controller in the app, no matter how simple the use.

The C# wrapper class is simply an artifact of how I originally defined the input interface. Had I defined that interface in a C++ WinRT extension library, the entire `GamepadInputService` could have been written in C++.

21.5.4 Adding from the code-behind

By now you should have the code-behind input service loading code memorized. The `GamepadInputService` addition follows the same pattern as all the others. The next listing has the updated `Loaded` event handler.

Listing 21.25 Updated `MainPage` `Loaded` event handler

```
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
  var vm = DataContext as MainViewModel;
  vm.InputServices.Clear();


  var keyInput = new KeyboardInputService();
  keyInput.Enable(null);

  var accelerometerInput = new AccelerometerInputService();
  accelerometerInput.Enable(null);

  var pointerInput = new PointerInputService();
  pointerInput.Enable(PlayField);

  var gamepadInput = new GamepadInputService();
  gamepadInput.Enable(null);

  vm.InputServices.Add(keyInput);
  vm.InputServices.Add(accelerometerInput);
  vm.InputServices.Add(pointerInput);
  vm.InputServices.Add(gamepadInput);
}
```

Create gamepad service

Add to collection

21.5.5 Compiling and deploying

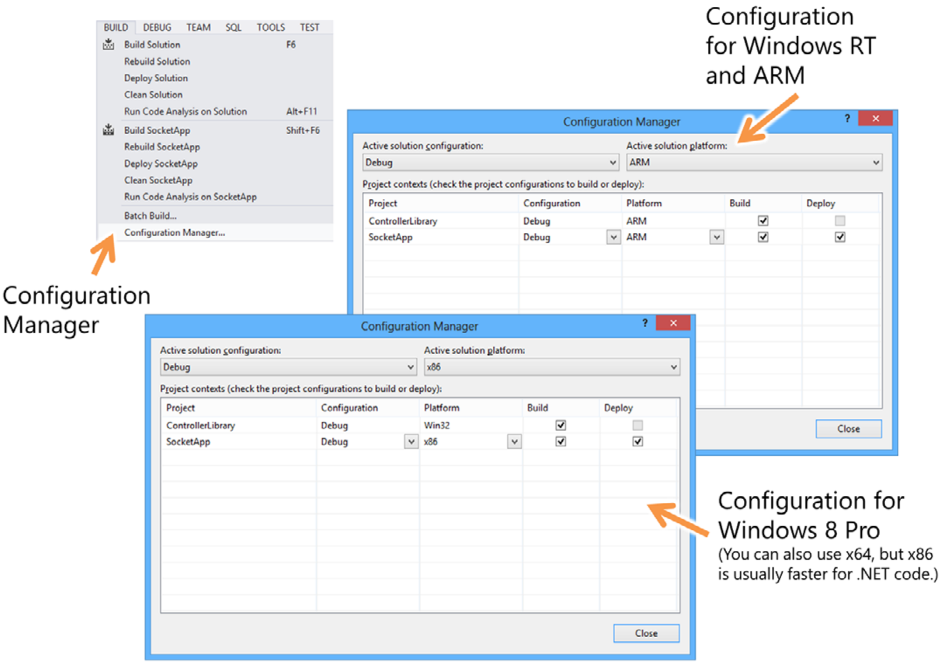
If you're compiling and deploying to the same x86 architecture you're developing on, this is business as usual. But if you're targeting another architecture, like ARM, you need to take extra steps.

Typically, C# apps are compiled to "any CPU," which means they can run on anything supported by WinRT XAML apps. The moment you include C++ code, you have to start creating architecture-specific builds and deployment packages.

One way to do this while debugging is to use Configuration Manager. By selecting the correct architecture from the Active Solution Platform drop-down list on the top right, you can deploy and debug on the remote ARM machine. Figure 21.13 shows what this process looks like.

It'll take practice to get used to how to do the deployments, especially if you work like I do. I always compile and start without debugging remotely to ARM and then start debugging locally on x86. In that way, I can run both endpoints of the app.

I love that I can use an Xbox controller on my Surface tablet. The USB port on the device may seem like such a simple feature, but it opens up a whole world of devices, storage mechanisms, and more.


BUILD
DEBUG
TEAM
SQL
TOOLS
TEST
Build Solution F6
Rebuild Solution
Deploy Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Build SocketApp Shift+F6
Rebuild SocketApp
Deploy SocketApp
Clean SocketApp
Run Code Analysis on SocketApp
Batch Build...
Configuration Manager...
Configuration Manager
?
×
Close
Configuration Manager
?
×
Close

Configuration for Windows RT and ARM

Active solution configuration: Debug

Active solution platform: ARM

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
ControllerLibrary	Debug	ARM	☑	☐
SocketApp	Debug	ARM	☑	☑

Configuration Manager

Active solution configuration: Debug

Active solution platform: x86

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
ControllerLibrary	Debug	Win32	☑	☐
SocketApp	Debug	x86	☑	☑

Configuration for Windows 8 Pro
(You can also use x64, but x86 is usually faster for .NET code.)

Figure 21.13 Configuration Manager showing the configuration for an ARM (Surface) deployment on the top and an x86 (desktop) configuration on the bottom. Select the proper configuration using the Active Solution Platform drop-down list.

Incorporating a C++ component into our app was actually quite easy. I'm not a C++ expert by any means, but I found it easy to work with and to pull into the solution. The new WinRT extension library project type takes care of all the fun behind the scenes that makes the types and methods visible and usable from our app. As someone who has worked a lot with COM and Win32 DLLs in the past, I find this new approach a breath of fresh air.

21.6 Summary

Most human input is handled by common controls like `Button`, `TextBox`, `GridView`, and others. For some apps, like ours, that built-in control input isn't sufficient, so we need to turn to lower-level input mechanisms.

Luckily, WinRT and C# together provide a rich set of APIs for getting user input from all of the common devices, plus sensors. A good practice when interfacing with something that has many possible implementations is to create a generic interface. For this app, you created a generic `IInputService` interface, and each concrete input device service class implemented that. Because of this, you can treat each input device identically, without worrying about the internal implementation.

The first input service you implemented using this interface is the keyboard input service. This made it easy to use the cursor keys on the keyboard to rotate and move the ship. The math to handle this was relatively simple, requiring only a couple of calculations.

From there, we looked at pointing devices. WinRT treats your finger, a mouse, and a pen all the same, so it was easy to use a single API to access all three of those devices. The API itself was quite simple to use, but the math you had to incorporate to track position was a bit more complex.

One nontraditional input device is the accelerometer. Most portable devices include an accelerometer, so having access to it can really liven up your app. WinRT XAML normally uses the accelerometer just for page orientation and rotation, but you don't need to stop there. In our app, you used the accelerometer to move the ship across the game field.

For times when the built-in input devices don't cover everything you need, you can turn to other means such as DirectX, if DirectX has access to the device through the XInput library. One such example is an Xbox gamepad. You can't use it directly from C#, but if you create a simple C++ WinRT extension assembly, you can consume that from C# and thereby access the device. Interestingly, you could then use that same library from a JavaScript app as well.

In the next chapter, we'll wrap up this app by adding support for saving app settings and for properly handling suspend and resume behaviors.

22

App settings and suspend/resume

This chapter covers

- Creating an app settings UI
- Using the `ApplicationData` class
- Using app settings
- Handling app suspend and resume

As we near the end of this book, there are a few important topics to cover that most app developers address later in their app development cycle. The first is the persistent storage of application settings. Most apps have some sort of secondary data, typically configuration information that they need to save. Windows 8 provides not only standardized classes for storing and retrieving this data but also a standardized approach to the UI for entering this data.

The second important topic is managing the app's lifetime. Many apps get away without ever dealing with app suspend and resume, but the experience just isn't that great when developers skip this. An app that suspends and then comes back looking like a brand-new launch, all because the user flicked over to the mail app for a second, is unlikely to be used very often.

Together, both of these will round out the knowledge you require to wrap up the code and UI for your app. These little details may seem unimportant compared to the rest of the app functionality, but given the broad (and often unforgiving) audience made available through the Windows Store, even the smallest detail can make or break an app.

In this chapter you'll learn how to create the appropriate app settings UI to manage those settings from within the app. Then, we'll look at how to use the `ApplicationData` class to save and load app settings. After all of this is in place, we'll look at something commonly associated with app settings: the app suspend and resume events.

22.1 App settings UI and architecture

In chapter 15, I covered the basics of working with files. This is useful when you have actual documents or other filesystem objects that your app creates or uses. Often, however, app settings are best represented as containers of values, or name-value pairs, rather than complete files. Silverlight developers may be familiar with this from the `IsolatedStorage` wrapper classes, which handled app settings this way.

When working with this type of data in Windows 8, the easiest approach is to use the `ApplicationData` class.

When working with these app settings, you'll typically need to provide an interface the user can use to maintain them. In desktop apps, this was typically under Tools > Options or Edit > Preferences, but that varied from application to application. Windows Store apps have a new standardized way to provide app settings using the Settings charm.

In this section, I'll show how to create the settings UI working against a stubbed-out settings class. Next, I'll show you how to create a UI for those settings. Then, I'll show you how to save and restore those settings both on the local machine as well as account-wide using the `ApplicationData` class.

22.1.1 Creating the settings infrastructure

Our network almost-game app doesn't have many configuration options. It does have two that are worth saving, however:

- If a server, whether it should autostart when the app is launched
- If a client, what IP address the app should try to connect to

I'll bet you wish we had implemented the first setting long ago. It gets tedious pressing the same app bar button every time the app launches during debug sessions.

These two settings are specific to the machine. If you log in as the same user on two machines, it probably wouldn't make sense for both of them to be the server or both of them to try to connect to the same IP address. For that reason, they'll be good for demonstrating local settings. But I also need to demonstrate roaming settings, so I'll add a third setting—an otherwise meaningless setting—specifically for testing and demonstrating that functionality.

When working with settings, the changes should be committed when the control itself changes or when the Settings pane is dismissed. Either approach is equally valid as long as you don't require a separate "save" interaction like a button.

In this app, you'll save all settings when the page is dismissed. This will still require a commit function in the `ServiceSettings` class you'll use. The stubbed-out code for the `SettingsService` class, without any actual saving or loading from the filesystem, is shown in the following listing. Create this new class named `SettingsService` in the Services folder.

Listing 22.1 The stubbed-out `SettingsService` class

```
using System;
using Windows.Storage;

namespace SocketApp.Services
{
  public class SettingsService
  {
    public static event EventHandler SettingsChangedRemotely;

    public static string RemotePlayerAddress { get; set; }
    public static bool AutoStartAsServer { get; set; }

    public static string SharedTestProperty { get; set; }

    public static void Save() {  }
    public static void Load() {  }
  }
}
```

SettingsChangedRemotely event

Local settings

Roaming setting

Persistence

The `SettingsChangedRemotely` event handler will be used when you integrate with the `ApplicationSettings` class and need to handle times when data is synchronized from a remote setting. The awkward name is so that it isn't confused with any event that may alert you to local data changes.

This class will provide the interface to the `ApplicationSettings` storage class. Although it could be used directly, because your settings are super simple, it's a good idea to follow the same viewmodel pattern you've used throughout this app. The next listing has the `SettingsOptionsViewModel` class you'll need to create in the ViewModel folder.

Listing 22.2 The Settings pane viewmodel and properties

```
using GalaSoft.MvvmLight;
using SocketApp.Services;
using System;

namespace SocketApp.ViewModel
{
  public class SettingsOptionsViewModel : ViewModelBase
  {
    public string RemotePlayerAddress
    {
      get { return SettingsService.RemotePlayerAddress; }
      set
      {
```

Bindable properties

```
        SettingsService.RemotePlayerAddress = value;
        RaisePropertyChanged(() => RemotePlayerAddress);
      }
    }

    public bool AutoStartAsServer
    {
      get { return SettingsService.AutoStartAsServer; }
      set
      {
        SettingsService.AutoStartAsServer = value;
        RaisePropertyChanged(() => AutoStartAsServer);
      }
    }

    public string SharedTestProperty
    {
      get { return SettingsService.SharedTestProperty; }
      set
      {
        SettingsService.SharedTestProperty = value;
        RaisePropertyChanged(() => SharedTestProperty);
      }
    }
  }
}
```

Bindable properties

The class is named `SettingsOptionsViewModel` because it applies just to the Options settings page. If you were to add a second settings page, say, Connectivity, you'd add a `SettingsConnectivityViewModel` to pair with it.

The viewmodel starts like any other viewmodel. But notice how the backing properties aren't local variables but are instead references to the static properties of the `SettingsService`. This approach may seem somewhat awkward, but it's helpful in that you don't need to keep track of multiple versions of the data, and it will always expose the latest values regardless of where they were changed from.

One hole in this approach, which isn't an issue for this app, is that other pages that change the settings using their own viewmodel and reference to the `SettingsService` don't trigger change notifications. You can address this by having the `SettingsService` also be an `ObservableObject`, but that would be overkill for this app.

One thing you do want to track is when changes are made externally, as the result of a roaming settings synchronization. The following listing includes the code that will enable this in the viewmodel (remember: you haven't yet implemented this in the `SettingsService` class).

Listing 22.3 Notification methods in `SettingsOptionsViewModel`

```
public SettingsOptionsViewModel()
{
  SettingsService.SettingsChangedRemotely +=
        OnSettingsChangedRemotely;
}
```

Wire up change notification

```
~SettingsOptionsViewModel()
{
  SettingsService.SettingsChangedRemotely -=
        OnSettingsChangedRemotely;
}

private void OnSettingsChangedRemotely(object sender, EventArgs e)
{
  NotifySettingsChanged();
}

private void NotifySettingsChanged()
{
  RaisePropertyChanged(() => RemotePlayerAddress);
  RaisePropertyChanged(() => AutoStartAsServer);
  RaisePropertyChanged(() => SharedTestProperty);
}
```

Unwire change notification

This code wires up the settings change notification with the `SettingsService` class. By doing this, you'll be able to react to external settings changes. Note also the `NotifySettingsChanged` method. This notifies the binding system of settings changes but doesn't discriminate between which settings have actually changed. If you have a lot of settings, you may want to be pickier about which events you fire off. But because settings are unlikely to change often, doing so would be more for peace of mind than for any real performance reason.

Local or roaming settings

There are two types of settings: local and roaming. Local settings never leave the machine. Roaming settings are synchronized across machines that share the same Microsoft account. This is how, for example, your start screen background and lock screen images get synchronized across multiple PCs.

When saving settings for your app, put some thought into what settings are machine-specific and what settings can be generalized to the overall account. For example, game-level completion and scores would be something that you may want to make available across all machines for this user. Settings specific to a machine configuration like accelerometer sensitivity or network connection information, however, should be considered local settings.

Rather than pushing this decision on the user through a configuration dialog, make intelligent decisions about how to store this information and where.

For more information, see http://bit.ly/WinRTRoamingData.

The final update to make to the viewmodel code is to enable saving and loading of settings. The next listing has the `Save` and `Load` methods to add to the `SettingsOptionsViewModel` class.

Listing 22.4 Persistence methods in `SettingsOptionsViewModel`

```
public void Save()
{
  SettingsService.Save();
}

public void Load()
{
  SettingsService.Load();
  NotifySettingsChanged();
}
```

Save settings

Load settings

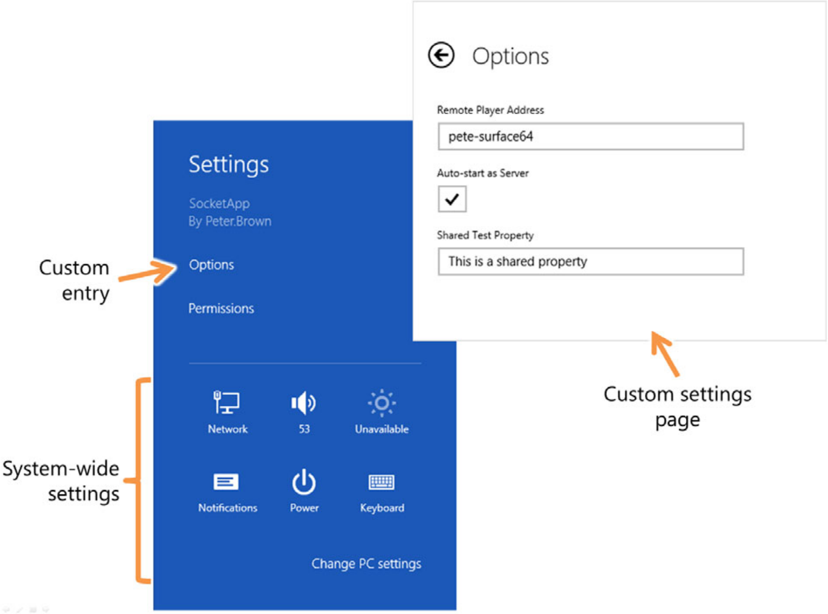
Notify binding system

With the stub of the `SettingsService` in place and a viewmodel established so you can talk with it from the UI, it's time to create the settings flyout.

22.1.2 Creating a settings UI

In Windows 8, apps are expected to have all persistent settings configured through the Settings charm. Your users will look for the settings here, so don't provide other mechanisms for displaying settings dialogs.

The settings UI is really easy to create. You provide a number of settings categories, and then when the user clicks one of them, Windows sends you a message telling you to display the settings UI. The UI display is a flyout that appears from the right and


Settings
SocketApp
By Peter.Brown
Options
Permissions
Network
53
Unavailable
Notifications
Power
Keyboard
Change PC settings
Options
Custom entry
System-wide settings
Custom settings page

Remote Player Address
pete-surface64

Auto-start as Server
☑

Shared Test Property
This is a shared property

Figure 22.1 The Settings pane for our socket-based multiplayer almost-game, with the standard page and our custom page (both visually truncated to fit the illustration). Clicking Options displays the custom options page. Clicking the back button on the options page brings back the main settings page. Also note how the custom options page is wider than the main settings page.

supports the “light dismiss” behavior. That is, clicking anywhere outside it will automatically dismiss the UI.

Figure 22.1 shows the Settings pane for this app, as well as the custom settings page you’ll create.

The settings page is implemented as a regular XAML page. Create one named SettingsOptionsPage, using the blank page template. The following is the markup for this new page.

Listing 22.5 The initial Settings pane SettingsOptionsPage.xaml

```
<Page
  x:Class="SocketApp.SettingsOptionsPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:SocketApp"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <UserControl.Resources>                                              Default
    <Style TargetType="TextBlock"                                      text style
           BasedOn="{StaticResource BodyTextStyle}">
      <Setter Property="Foreground" Value="Black" />
    </Style>
                                                              Back button
    <Style x:Key="SettingsBackButtonStyle"                    style
           TargetType="Button">
      <Setter Property="MinWidth" Value="0" />
      <Setter Property="Margin" Value="20,0,0,0" />
      <Setter Property="VerticalAlignment" Value="Bottom" />
      <Setter Property="FontFamily" Value="Segoe UI Symbol" />
      <Setter Property="FontWeight" Value="Normal" />
      <Setter Property="FontSize" Value="26.66667" />
      <Setter Property="AutomationProperties.AutomationId"
              Value="BackButton" />
      <Setter Property="AutomationProperties.Name" Value="Back" />
      <Setter Property="AutomationProperties.ItemType"
              Value="Navigation Button" />
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="Button">
            <Grid x:Name="RootGrid" Width="36" Height="36"
                  Margin="-3,0,7,0">
              <Grid Margin="-1,-1,0,0">
                <TextBlock x:Name="BackgroundGlyph"
                           Text="&#xE0D4;"
                           Foreground="White" />
                <TextBlock x:Name="NormalGlyph"
                           Text="{StaticResource BackButtonSnappedGlyph}"
                           Foreground="Black" />
              </Grid>

              <VisualStateManager.VisualStateGroups>
                <VisualStateGroup x:Name="CommonStates">
```

```
                  <VisualState x:Name="Normal" />
                  <VisualState x:Name="PointerOver" />
                  <VisualState x:Name="Pressed" />
                  <VisualState x:Name="Disabled" />
                </VisualStateGroup>
                <VisualStateGroup x:Name="FocusStates">
                  <VisualState x:Name="Focused" />
                  <VisualState x:Name="Unfocused" />
                  <VisualState x:Name="PointerFocused" />
                </VisualStateGroup>
              </VisualStateManager.VisualStateGroups>
            </Grid>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>

  </UserControl.Resources>

  <Grid Background="White">
    <Grid.RowDefinitions>
      <RowDefinition Height="80" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid Grid.Row="0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>

      <Button Click="OnSettingsBackButtonClick"              Back button
              Grid.Column="0"
              Style="{StaticResource SettingsBackButtonStyle}"/>

      <TextBlock Margin="10,0,0,7"                             Page title
                 Grid.Column="1"
                 VerticalAlignment="Bottom"
                 Style="{StaticResource SnappedPageHeaderTextStyle}"
                 Text="Options"
                 Foreground="Black" />
    </Grid>

    <Grid Grid.Row="1" Margin="30"
          x:Name="ContentPanel">
      <TextBlock Text="Dummy text" />                   Content
    </Grid>                                             will go here

  </Grid>
</Page>
```

I prefer to name each settings page with the “Settings” prefix to distinguish it from other pages in the app. Another approach would be to put them in a separate folder/ namespace named SettingsPages or similar.

This page has no real content, just a back button and title and the styles to support them. To create the back button style, I copied the snapped back button template from App.xaml and made some changes to it. For simplicity here, I removed the various visual states. In your own app, you may wish to retain them for hover and press at a minimum.

The code-behind for this page needs to wire up the viewmodel (I didn't use the locator in this case, because I'm referring to the viewmodel from code-behind anyway) and also handle the back button `Click` event. The following listing has the code you need.

Listing 22.6 The Settings pane code-behind

```
using SocketApp.ViewModel;
using System;
using Windows.UI.ApplicationSettings;
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Navigation;

namespace SocketApp
{
  public sealed partial class SettingsOptionsPage : Page
  {
    private SettingsOptionsViewModel _vm;
    public SettingsOptionsPage()
    {
      this.InitializeComponent();

      _vm = new SettingsOptionsViewModel();
      DataContext = _vm;

      this.Unloaded += SettingsOptionsPage_Unloaded;
    }

    void SettingsOptionsPage_Unloaded(object sender, RoutedEventArgs e)
    {
      if (_vm != null)
        _vm.Save();
    }

    private void OnSettingsBackButtonClick(object sender,
                                           RoutedEventArgs e)
    {
      var parent = this.Parent as Popup;
      if (parent != null)
        parent.IsOpen = false;

      if (ApplicationView.Value != ApplicationViewState.Snapped)
        SettingsPane.Show();
    }
  }
}
```

Create viewmodel

Wire up unload event

Save on unload

Close parent popup

Show main Settings pane

There are two new and important pieces in this code-behind. The first is how to save the settings when the page is unloaded. You do this here in order to handle both the back button and light dismiss approaches to closing the page. The second is how the back button works. The back button closes the parent popup and then, if the app isn't in snapped state, displays the main Settings pane.

The `SettingsPane.Show` method is usable anywhere in the app. For example, if you want to have an app bar button to display settings, its click event would simply contain this code. Another common location to do this is in an app like a game where you're walking the user through a number of things and will need to programmatically open the Settings pane for them.

The code to display the Settings pane in the code-behind is simple, but that's only because all the details are wrapped into another service class. In the Services folder, create a new class named `SettingsPaneService` using the code in the following listing.

Listing 22.7 The `SettingsPaneService` class

```
using System;
using Windows.UI.ApplicationSettings;
using Windows.UI.Popups;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Media.Animation;

namespace SocketApp.Services
{
  public class SettingsPaneService : IDisposable
  {
    public SettingsPaneService()
    {
      SettingsPane.GetForCurrentView().CommandsRequested +=
                OnSettingsPaneCommandsRequested;
    }

    ~SettingsPaneService()
    {
      if (!_isDisposed)
        Dispose();
    }

    private bool _isDisposed = false;
    public void Dispose()
    {
      SettingsPane.GetForCurrentView().CommandsRequested -=
                OnSettingsPaneCommandsRequested;

      _isDisposed = true;
      GC.SuppressFinalize(this);
    }

    private void OnSettingsPaneCommandsRequested(SettingsPane sender,
                    SettingsPaneCommandsRequestedEventArgs args)
    {
    }

  }
}
```

CommandsRequested event

Clean up event handler

Command handling goes here

This class registers for the key `CommandsRequested` event on startup. This event is what Windows will use to get a list of commands from you when the user opens the Settings pane.

Like many of the core WinRT notification events, it's important that you have only a single active listener on the `CommandsRequested` event. To ensure this, I implemented the dispose pattern and made sure the event is unwired when this class is disposed or is collected.

Listing 22.7 doesn't have all the code. In particular, the callback for handling the command requests is empty. The following listing has the remaining implementation code.

Listing 22.8 The main functions in the `SettingsPaneService` class

```
private const int SettingsPage1ID = 1001;
private void OnSettingsPaneCommandsRequested(SettingsPane sender,
                        SettingsPaneCommandsRequestedEventArgs args)
{
  var options = new SettingsCommand(
          SettingsPage1ID, "Options",
          OnSettingsOptionsCommand);

  args.Request.ApplicationCommands.Add(options);
}

private Popup CreateSettingsPopup(Page settingsPage)
{
  var popup = new Popup();

  var fullWidth = Window.Current.CoreWindow.Bounds.Width;
  var fullHeight = Window.Current.CoreWindow.Bounds.Height;

  var paneWidth = 646;

  popup.IsLightDismissEnabled = true;
  popup.Child = settingsPage;

  settingsPage.Height = fullHeight;
  settingsPage.Width = paneWidth;

  var transition = new PaneThemeTransition();

  if (SettingsPane.Edge == SettingsEdgeLocation.Right)
  {
    Canvas.SetLeft(popup, fullWidth - paneWidth);

    transition.Edge = EdgeTransitionLocation.Right;
  }
  else
  {
    Canvas.SetLeft(popup, 0);

    transition.Edge = EdgeTransitionLocation.Left;
  }
```

Create settings command

Add settings command

Create popup

Pane width up to you

Enable light dismiss

Create transition (animation)

Set transition edge

```
  popup.ChildTransitions = new TransitionCollection();
  popup.ChildTransitions.Add(transition);

  return popup;
}

private void OnSettingsOptionsCommand(IUICommand command)
{
  var options = new SettingsOptionsPage();

  var popup = CreateSettingsPopup(options);
  popup.IsOpen = true;
}
```

Add transition

The remaining code in this class handles creating the settings command (you could have multiple, but I have only one) and the settings popup. Your Settings pane can be any width you want from full screen (like the current version of the Windows Store app) to almost nothing. I prefer to have a slightly wider settings page so things don't feel cramped. The UI guidelines suggest using either narrow (346 pixels) or wide (646 pixels), but this isn't enforced.

This code also uses the popup class to host the settings page you've created. The popup has the "light dismiss" behavior enabled and also uses some of the built-in entrance animations to make it slide out from the edge. Those animations, known as transitions, are common across the OS and very highly optimized, so it's a good idea to use them instead of creating your own.

There's a little more code you need to write to create the `SettingsPaneService`. In the MainPage.xaml.cs file, add the code from the next listing. You'll likely already have an empty `OnNavigatedTo` handler, so simply replace it.

Listing 22.9 Wiring up Settings pane handling in MainPage.xaml.cs

```
private SettingsPaneService _settingsPaneService;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
  if (_settingsPaneService == null)
    _settingsPaneService = new SettingsPaneService();
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
  if (_settingsPaneService != null)
  {
    _settingsPaneService.Dispose();
    _settingsPaneService = null;
  }
}
```

Wire up handlers

Unwire handlers

Notice how this code disposes of the `SettingsPaneService` instance when the page is navigated away from. This will clean up that event handler in case you want to show settings from another page.

TIP For more information on the `SettingsPane` class, see this page on MSDN: http://bit.ly/WinRTSettingsPane. For guidelines on settings in general and the UX of the Settings pane, see http://bit.ly/WinRTSettingsGuide.

If you run the app right now and invoke the Settings charm, you'll see your entry and the page associated with it. The page doesn't actually interact with the stored settings, however. For that, you'll use the markup in the following listing. Replace the `ContentPanel Grid` in that markup with this listing.

Listing 22.10 SettingsOptionsPage.xaml content markup

```
<Grid Grid.Row="1" Margin="30"
      x:Name="ContentPanel">
  <StackPanel>
    <StackPanel Margin="0,0,0,10" Width="350"
                HorizontalAlignment="Left">
      <TextBlock Text="Remote Player Address"
                 Style="{StaticResource CaptionTextStyle}"
                 Foreground="Black"/>
      <TextBox Text="{Binding RemotePlayerAddress, Mode=TwoWay}"
               Margin="0,10,0,0"
               BorderBrush="DarkGray"/>
    </StackPanel>

    <StackPanel Margin="0,0,0,10" Width="350"
                HorizontalAlignment="Left">
      <TextBlock Text="Auto-start as Server"
                 Style="{StaticResource CaptionTextStyle}"
                 Foreground="Black" />
      <CheckBox IsChecked="{Binding AutoStartAsServer, Mode=TwoWay}"
                Margin="0,10,0,0"
                Foreground="Black"
                BorderBrush="#FFBFBFBF"
                BorderThickness="2"
                IsThreeState="False"
                HorizontalAlignment="Left"/>
    </StackPanel>

    <StackPanel Margin="0,0,0,10" Width="350"
                HorizontalAlignment="Left">
      <TextBlock Text="Shared Test Property"
                 Style="{StaticResource CaptionTextStyle}"
                 Foreground="Black" />
      <TextBox Text="{Binding SharedTestProperty, Mode=TwoWay}"
               Margin="0,10,0,0"
               BorderBrush="DarkGray" />
    </StackPanel>
  </StackPanel>
</Grid>
```

Remote player address property

Autostart server property

Shared test property

If you run the app now, you'll see the settings UI controls on the settings page. Nothing is persisted to the application settings storage just yet, but you can test the binding by invoking the Settings charm, opening the page, making some changes, and closing the page. If the settings are there when you open the page again in the same app session, you'll know everything is working.

A settings page is like any other page in the app, but it's contained in a popup and displayed only when requested from Windows. As an app developer, you tell Windows which settings commands you offer, and Windows will let the user choose what to display. This decoupling is nice because it allows you to seamlessly plug into the standard Settings pane while giving you, the app developer, the freedom to design your settings UI any way you want.

In the next section, we'll wire up the UI and service class to the WinRT app settings infrastructure.

22.2 Persisting and using settings

It seems obvious, but settings are only useful if they are saved, and even then, only if they are also loaded. Right now, your app is making all of the settings changes in memory but is not persisting them to the settings stores.

Recall that there are two different settings locations that your app will use:

- Local
- Roaming

All settings are per user, so local settings are per user on a single machine. Roaming settings are per user across multiple machines. The user must be using the same Microsoft account for the changes to roam from machine to machine.

In this section you'll modify the code in this app to both load and save settings using the `ApplicationData` class. You'll then take this data and modify the `MainPageViewModel` to use it for the user's benefit.

22.2.1 Loading and saving settings values

There are as many ways to store settings and configuration as there are programmers building apps. I encourage you, however, to use the standard `ApplicationData` class built into WinRT when building Windows 8 apps. By doing so, you can take advantage of the support built into Windows 8 for roaming synchronization, per-user storage, and more.

The next listing shows the `SettingsService` updated to make use of the `ApplicationData` class.

Listing 22.11 `SettingsService` updates to load and save settings

```
using System;
using Windows.Storage;

namespace SocketApp.Services
{
  public class SettingsService
  {
    public static event EventHandler SettingsChangedRemotely;
public static string RemotePlayerAddress { get; set; }
public static bool AutoStartAsServer { get; set; }
public static string SharedTestProperty { get; set; }

public SettingsService()
{
  ApplicationData.Current.DataChanged += OnDataChanged;
}
```

Wire up data changed event

```
~SettingsService()
{
  ApplicationData.Current.DataChanged -= OnDataChanged;
}
```

Unwire data changed event

```
private static void OnDataChanged(ApplicationData sender, object args)
{
  Load();

  RaiseDataChanged();
}
```

Reload data when changed

```
private static void RaiseDataChanged()
{
  if (SettingsChangedRemotely != null)
    SettingsChangedRemotely(null, EventArgs.Empty);
}
```

Property keys

```
private const string RemotePlayerAddressKey = "RemotePlayerAddress";
private const string AutoStartAsServerKey = "AutoStartAsServer";
private const string SharedTestPropertyKey = "SharedTestProperty";
```

Save values

```
public static void Save()
{
  var current = ApplicationData.Current;

  current.RoamingSettings.Values[SharedTestPropertyKey] =
           SharedTestProperty;

  current.LocalSettings.Values[RemotePlayerAddressKey] =
           RemotePlayerAddress;
  current.LocalSettings.Values[AutoStartAsServerKey] =
           AutoStartAsServer;
}
```

Get roaming settings value

```
private static T GetRoamingValue<T>(string key, T defaultValue)
{
  var current = ApplicationData.Current;

  if (current.RoamingSettings.Values.ContainsKey(key))
    return (T)(current.RoamingSettings.Values[key]);
  else
    return defaultValue;
}
```

Get local settings value

```
private static T GetLocalValue<T>(string key, T defaultValue)
{
  var current = ApplicationData.Current;
```

```
      if (current.LocalSettings.Values.ContainsKey(key))
        return (T)(current.LocalSettings.Values[key]);
      else
        return defaultValue;
    }

    public static void Load()
    {
      RemotePlayerAddress =
         GetLocalValue<string>(RemotePlayerAddressKey, string.Empty);
      AutoStartAsServer =
         GetLocalValue<bool>(AutoStartAsServerKey, false);

      SharedTestProperty =
         GetRoamingValue<string>(SharedTestPropertyKey, string.Empty);
    }
  }
}
```

Load settings values

To save a value to the settings store, it's sufficient to update its value in the `ApplicationData` class. This is done through either the `LocalSettings.Values` or `RoamingSettings.Values` dictionary. Each entry in the settings is specified by a key of your choice. I recommend keeping them in constants so you don't end up with a typo ruining your day.

The only real complexity in this code is there to ensure that null and missing settings are handled gracefully. For that reason, I added the `GetRoamingValue` and `GetLocalValue` functions. Those two functions first check for the existence of a value before returning it or a specified default.

Where will the settings first be loaded? You'll do that in the app.xaml `OnLaunched` handler. At the very first line in the `OnLaunched` method in App.xaml.cs, simply add in the call to the `Load` method as shown here.

Listing 22.12 App.xaml.cs update to load settings

```
using SocketApp.Services;
...
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
  SettingsService.Load();

  ...
}
```

Load settings first

Rest of launch code

This single line of code makes the settings available throughout the app. Make some changes, save the settings, and then close and reopen the app to make sure they stuck.

Passwords

Passwords should never be stored using the normal local and roaming data stores in the `ApplicationData` class. Instead, passwords should be stored using `PasswordVault` and the credential locker.

(continued)

The credential locker, available through the `ApplicationData` WinRT class, provides a secure encryption location to store sensitive data, specifically passwords and their related properties. Each credential, stored as a `PasswordCredential` instance, contains the password, properties of the password, the associated resource, and the associated username. If your app connects to remote services using any sort of credentials, it's best to store them here.

For more information, see the `PasswordVault` class documentation on MSDN: http://bit.ly/WinRTPasswordVault.

If you have more than one PC *running with the same Microsoft account*, then you can test the roaming settings. (If you don't have access to two machines, you'll just need to trust me, because roaming settings only make sense in a multiple-machine scenario.) Deploy the app to both machines, and then debug on one of them. Set a breakpoint in the `SettingsService` in the `OnDataChanged` method. Then, change the single roaming setting on the second PC. Wait, and depending upon your connectivity (it may take several minutes), you'll see the change come through on the local PC.

The final thing you need to do with the settings is act on them elsewhere in the code.

22.2.2 Acting on the options

There are two options you want to use in your app. The first is the default remote player address. The second is whether or not to autostart as a server. You have the code to load and save the settings, but you don't have anything to actually use them. Let's fix that now.

Both of these changes can be made in the constructor of `MainViewModel`. Open that file and where you see the hardcoded server address, add in a reference to the `SettingsService`. Also, in the last line of the constructor, check the `AutoStartAsServer` property and call `Listen` as appropriate. The following listing shows the changes in context.

Listing 22.13 Updates to `MainViewModel` to use the stored options

```
public MainViewModel()
{
  ...

  //ServerAddress = "pete-surface64";
  ServerAddress = SettingsService.RemotePlayerAddress;

  ...

  if (SettingsService.AutoStartAsServer)
    Listen();
}
```

Set remote player address

Start server as last step in constructor

With just this code in place, the app doesn't respond to actual changes to settings during the app's runtime. So, setting the connection information and then trying to connect won't result in the expected outcome. To fully support this, you'll need to add more change notification from the `SettingsService` class and listen to that event here in the `MainViewModel`. This change is easy enough to add following the existing patterns. In fact, the code available online from www.manning.com/pbrown3 has this addition implemented.

Application settings are persisted by setting their values in a settings dictionary—there's no additional step. The synchronization of roaming settings is handled automatically by Windows, and the storage of local settings to disk is also automatically handled. There's no excuse not to use these built-in classes for storing your app's settings and state.

Now that you know how to save and maintain the app settings, let's look at how to respond to the app's lifetime changes—specifically suspend and resume.

22.3 Suspend and resume

In the past, the only application lifetime events developers had to think about were the shutdown messages sent from Windows or when the user closed the app via a menu or button. Maybe the developer also hooked into the activation messages to know when their app was brought to the foreground.

In Windows 8, app lifetime is a bit more complex, because Windows manages the life of the app based on what the user is doing and what resources are available. The end user controls when an app is suspended, but the OS typically controls when the app is shut down. I say "typically" because the user can use Alt-F4 or can swipe down from the top to close an app completely. But in my unscientific observation of my two children and my wife, no one actually does that; they simply hit the Windows key to get back to the Start page. Granted, my sample size is small.

When a suspended app is brought back to the foreground (either as the only app or as a snapped or filled app), it is said to have been "resumed." To the user, a resumed app should look as if it were never stopped, with the exception of perhaps a "paused" display for a game. The app should pick up where it left off.

This suspension process allows Windows to make the most of processing time and memory on low-power devices such as the ARM tablets targeted by Windows RT.

In this section, we'll look at suspend and resume in the context of the app we've developed over the past several chapters. First, we'll look at how to successfully suspend the app, including any state saving or other cleanup, and then look at the code to resume from suspension.

22.3.1 Suspending your app

Windows 8 allows up to two Windows Store apps to be visible at any one time. When your app isn't visible but wasn't explicitly closed, it's suspended. Suspended apps are retained in memory (as long as there's sufficient memory) but are blocked from running. Nevertheless, it's good form to stop any background threads on suspend, to ensure a clean termination.

Ideally, your app should save important state as it changes, on a timed schedule, or after major events (like level completion during a game). That way it doesn't matter when an app shuts down, because the latest state information is always up to date. If an app does this, saving state when suspended becomes unnecessary.

But, at the very least, the app should save state when it becomes suspended. The state needs to include everything the app needs to come up and look like it never went away. That may include which page the user was on, the values typed into fields on the form (even if they haven't yet been validated), any downloaded information, and more. It also needs to be sufficient to allow for cases when the app is never resumed but is instead loaded anew in the future.

Background and potentially longer-running asynchronous operations must be completed using the deferral pattern. You won't have forever, though. If the state saving takes a long time, you may be out of luck, because the app likely won't pass certification. For that reason, again, it's best to save state while the app is running.

The next listing shows the use of the deferral pattern in the `OnSuspending` method in app.xaml.cs.

Listing 22.14 The deferral pattern and suspending

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
  var deferral = e.SuspendingOperation.GetDeferral();


  deferral.Complete();
}
```

Suspending event

Get the deferral

Do work

Complete the deferral

The deferral pattern enables you to make asynchronous calls from within the event handler and defer app suspension until the `Complete` method is called. It's a good idea to use this pattern by default in the `OnSuspending` method.

Our app has no running state, but inside this method you'd use the `ApplicationData` class, or another method, to store things like the page the user is on, items that are selected in a list, current score and level, and the like. Don't store things that are worthless when the app is resumed. For example, you wouldn't want to store the location of the other player because that would be irrelevant when the app is brought back up.

22.3.2 Resuming activity

When the app comes back to life after a suspend operation, you'll need to reload all the state information previously saved. You'll also need to restart any background operations, start sending network messages again, and so on.

This is simply done by handling the appropriate activation kinds in the `OnLaunch` method in App.xaml.cs.

But if it uses the same launch method, how do you know if the app was shut down? This is provided in the `LaunchActivedEventArgs` instance passed in. Inspect the value of `PreviousExecutionState` and load settings or perform other operations as appropriate. This enumeration has five values, as shown in table 22.1.

Table 22.1 The possible values of the `PreviousExecutionState` property of the `LaunchActivatedEventArgs`

Value	Description
`NotRunning`	The app was simply not running.
`Running`	The app was running and not suspended.
`Suspended`	The app was suspended and is now being activated.
`Terminated`	The app was suspended and then the user's session closed normally. (They logged out of Windows, shut down the machine, and so on.)
`ClosedByUser`	The app was closed with Alt-F4, or a swipe down from the top, or manually from the list of running apps.

In this case, if the app is anything but `Running`, you should load state and settings. State might include things like what page the user was on, which items in a list were selected, and so on. Your app has no state, but it does have settings. The following listing includes the updated `OnLaunched` method with this check in place.

Listing 22.15 `UpdatedOnLaunched` method that handles settings loading

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
  Frame rootFrame = Window.Current.Content as Frame;

  if (args.PreviousExecutionState != ApplicationExecutionState.Running)
  {
    SettingsService.Load();
  }

  if (rootFrame == null)
  {
    rootFrame = new Frame();

    if (args.PreviousExecutionState !=
            ApplicationExecutionState.Terminated)
    {
    }

    Window.Current.Content = rootFrame;
  }
```

Check that it's not already running

Load settings

Default check for termination

```
  if (rootFrame.Content == null)
  {
    if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
    {
      throw new Exception("Failed to create initial page");
    }
  }
  Window.Current.Activate();

  DispatcherHelper.Initialize();
}
```

The only new code in this method is the execution state check near the top of the method. Earlier in this chapter, the `SettingsService.Load` method was put here, but without the benefit of this check.

> **NOTE** The current default project templates include a check for `ApplicationExecutionState.Terminated`, saying you need to reload state for all but that case. For many uses, this isn't really correct. You may want to load state when suspended or terminated but not load it when the app is running or closed by the user. Think through your app state and pick the option that makes sense for your use cases.

Application state can be more than just settings. It's about making the user feel like the app was still there, behind the current app, even though it may have been put in suspension. To do this correctly requires tracking a lot of state for the app and persisting it throughout the lifetime, but doing a final push on app suspend.

Your app doesn't have any state data, although it could persist store and connection information if that makes sense. Certainly a single-player local app would persist score, ship position, and more.

Regardless of how you treat app state or app settings, Windows notifies you of the lifetime of the app through the `OnLaunched` and `OnSuspending` methods. Through these two methods you can load and save whatever you need to ensure the best experience for your users. For more information on launching, suspending, and resuming apps, see http://bit.ly/WinRTAppLifetime.

22.4 Summary

Windows 8 provides a new standardized approach for the UI for app settings. There's a common location for the storage and a common class to use to maintain settings (`ApplicationSettings`), regardless of whether they're local or roaming. The settings UI is something you have control over, but the approach for displaying it is also standardized in the Settings pane.

Application settings and application state are important pieces of data that the app needs to maintain throughout its lifetime. Save your app state as the app is running, however, and don't simply wait for app suspension. This will not only save time, but it will also enable better recovery from crashes, dead batteries (on tablets/laptops), and tripped-over power cords on desktops. App settings should be saved immediately in the Settings pane.

With the new app lifetime model in Windows 8, it's essential to understand how to use the suspend and resume notifications to manage your app. Slow app suspend or a poor transition when resuming both really stand out to app users and to the app certification reviewers.

In the next chapter, we'll wrap up the book with a look at that final step before your users get the app: deployment and the Windows Store.

23

Deploying and selling your app

This chapter covers

- Testing for certification
- Sideloading for testing
- Listing in the Windows Store
- Enabling trial mode

In most development platforms we write about, the topic of deployment is more a discussion of the mechanical process of building a setup package of some type, checking dependencies and framework versions, putting the download on a website, and then letting people have at it. For Windows Store apps, the deployment process is both simpler and more important.

We're not going to talk much about coding or design (well, just a little) in this final chapter but more about getting your app out in front of your audience through the Windows Store. With the exception of the Windows Phone, this is new territory for Microsoft and for Microsoft developers; we've never before had a proper unified purchasing location for apps on our PCs. We've also never had the kind of opportunity that the Windows Store represents; getting an app into a retailer's product catalog was just not possible for most individuals and small companies, and one-off sales through personal websites rarely panned out.

In this chapter you'll first learn how to test the app against the Windows Store performance, API, and other runtime criteria using the Windows App Certification Kit (WACK) tool. Then, because apps should be tested, no matter how simple, we'll look at how to provide install packages for your test group to use to run your app, without involving the Windows Store. Then, before we look at how to list the app in the Windows Store, we'll take a quick peek at how to increase your sales by enabling trial mode.

23.1 Testing for certification

The process of submitting your app to the Windows Store doesn't start when the app is complete; it starts long before that, during your own internal testing and development cycle, when you run the WACK tool.

WACK contains a subset of the certification process used by the Windows Store. For an app to proceed through to some of the other internal Windows Store checks, it must first pass the WACK test. Because this is an automated test and can be run by developers, Microsoft has provided it as a standalone tool, installed as part of the development tools install. Figure 23.1 shows WACK installed and present on the Start page.

WACK will only test apps that have been installed on the PC. When testing your under-development app on the development PC, ensure you've built and deployed it locally first, or else you may test against an older version. This is especially true if you're deploying and debugging in the Simulator instead of the main PC.

Double-click the WACK icon to launch the app. It will require administrator privileges and will run as a desktop app. When it first comes up, you'll be prompted to pick a type of app to test, as shown in figure 23.2.

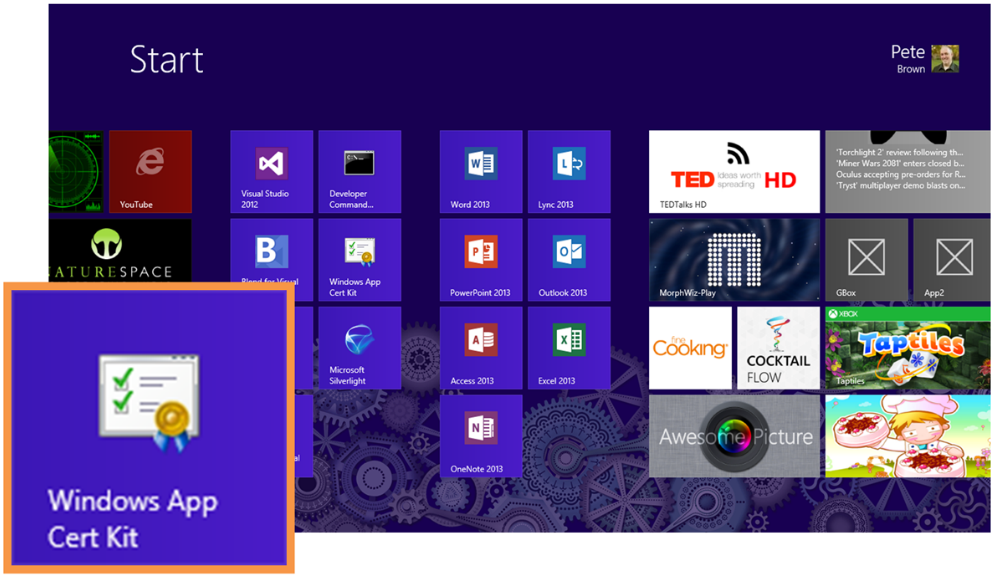
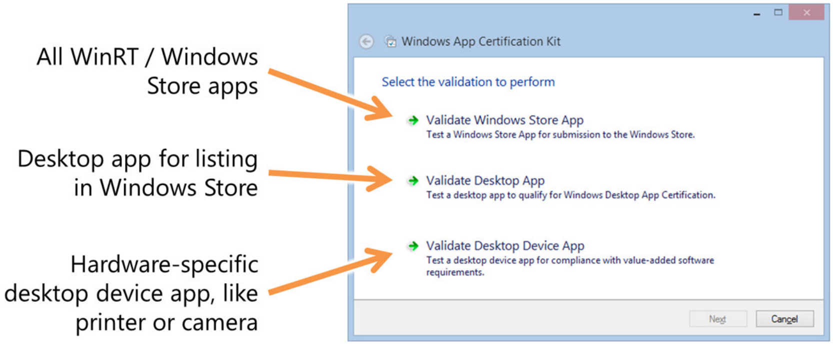

Start
Pete
Brown
YouTube
Visual Studio 2012
Developer Command...
Word 2013
Lync 2013
TEDTalks HD
Windows App Cert Kit
PowerPoint 2013
Outlook 2013
MorphWiz-Play
GBox
App2
Microsoft Silverlight
Access 2013
Excel 2013
COCKTAIL FLOW
Taptiles
OneNote 2013
Awesome Picture
Windows App
Cert Kit

Figure 23.1 The WACK tool is installed with the Windows Store app development tools.



All WinRT / Windows Store apps

Desktop app for listing in Windows Store

Hardware-specific desktop device app, like printer or camera

Windows App Certification Kit

Select the validation to perform

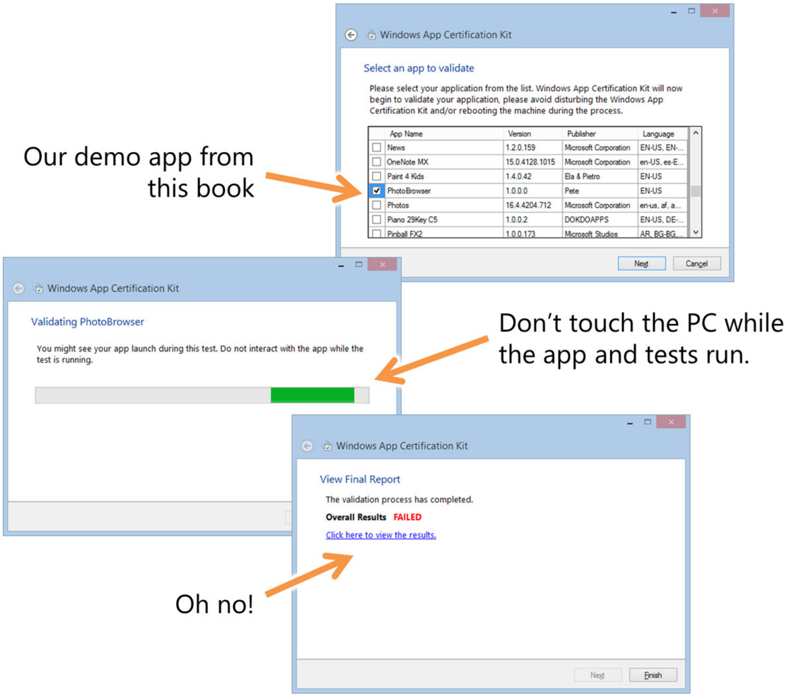
Validate Windows Store App
Test a Windows Store App for submission to the Windows Store.

Validate Desktop App
Test a desktop app to qualify for Windows Desktop App Certification.

Validate Desktop Device App
Test a desktop device app for compliance with value-added software requirements.

Figure 23.2 The first page of the WACK tool prompts you to pick the type of app to test. WinRT XAML apps will be the first option.

Once you select the first option, you'll be prompted to pick an installed app. As long as you have a valid developer license, you can run the WACK test against any app in the system; it doesn't reveal any sensitive data. Figure 23.3 shows the WACK test run against the PhotoBrowser app created across several earlier chapters.



Our demo app from this book

Windows App Certification Kit

Select an app to validate

Please select your application from the list. Windows App Certification Kit will now begin to validate your application, please avoid disturbing the Windows App Certification Kit and/or rebooting the machine during the process.

	App Name	Version	Publisher	Language
☐	News	1.2.0.159	Microsoft Corporation	EN-US, EN-...
☐	OneNote MX	15.0.4128.1015	Microsoft Corporation	en-US, es-E...
☐	Paint 4 Kids	1.4.0.42	Ela & Pietro	EN-US
☑	PhotoBrowser	1.0.0.0	Pete	EN-US
☐	Photos	16.4.4204.712	Microsoft Corporation	en-us, af, a...
☐	Piano 29Key C5	1.0.0.2	DOKDOAPPS	EN-US, DE-...
☐	Pinball FX2	1.0.0.173	Microsoft Studios	AR, BG-BG,...

Don't touch the PC while the app and tests run.

Windows App Certification Kit

Validating PhotoBrowser

You might see your app launch during this test. Do not interact with the app while the test is running.

Windows App Certification Kit

View Final Report

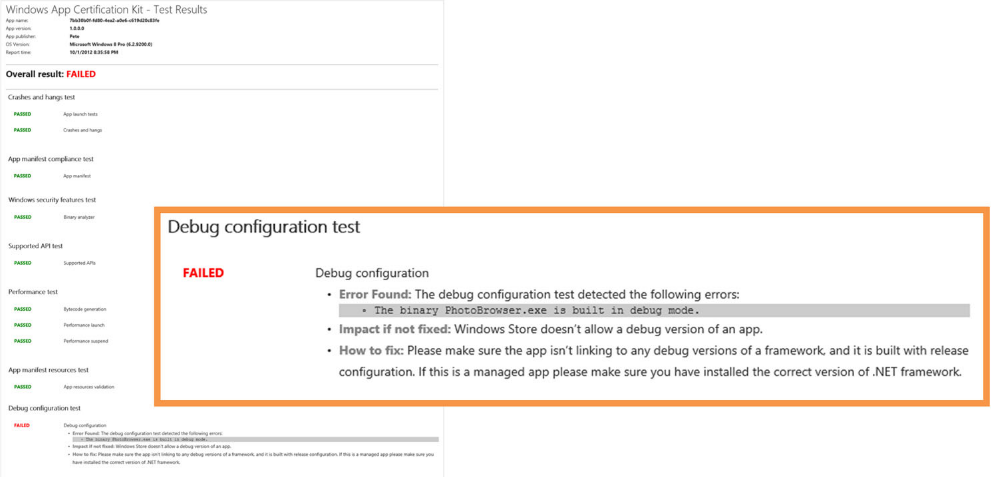
The validation process has completed.

Overall Results FAILED

Click here to view the results.

Oh no!

Figure 23.3 The second page of the WACK tool prompts you for the specific app to test. From there, click Next, and the test will run. This can take several minutes, especially for a complex app; you'll need to leave the app and PC alone during that time, because performance and other metrics will be captured. Finally, you'll see the results of the test. It failed, as you can see. Explanation is in the text.



Windows App Certification Kit - Test Results

App name: 7bb30b0f-fd80-4ea2-a0e6-c619d20c83fe
App version: 1.0.0.0
App publisher: Pete
OS Version: Microsoft Windows 8 Pro (6.2.9200.0)
Report time: 10/1/2012 8:35:58 PM

Overall result: FAILED

Crashes and hangs test	
PASSED	App launch tests
PASSED	Crashes and hangs

App manifest compliance test	
PASSED	App manifest

Windows security features test	
PASSED	Binary analyzer

Supported API test	
PASSED	Supported APIs

Performance test	
PASSED	Bytecode generation
PASSED	Performance launch
PASSED	Performance suspend

App manifest resources test	
PASSED	App resources validation

Debug configuration test

FAILED Debug configuration

- Error Found: The debug configuration test detected the following errors:
  - `The binary PhotoBrowser.exe is built in debug mode.`
- Impact if not fixed: Windows Store doesn't allow a debug version of an app.
- How to fix: Please make sure the app isn't linking to any debug versions of a framework, and it is built with release configuration. If this is a managed app please make sure you have installed the correct version of .NET framework.

Debug configuration test

FAILED Debug configuration

- **Error Found:** The debug configuration test detected the following errors:
  - `The binary PhotoBrowser.exe is built in debug mode.`
- **Impact if not fixed:** Windows Store doesn't allow a debug version of an app.
- **How to fix:** Please make sure the app isn't linking to any debug versions of a framework, and it is built with release configuration. If this is a managed app please make sure you have installed the correct version of .NET framework.

Figure 23.4 The test results from WACK. The PhotoBrowser app failed because it was compiled in debug configuration, not release.

Once the test completes, you'll be prompted to save the XML file with the results. You can then click the link to view them in Internet Explorer. In my case, the test failed and for a very good reason: The app was compiled in debug configuration rather than release. Figure 23.4 shows the results.

WACK will rate the app as Passed, Passed with Warnings, or Failed.

- *Failures* must be addressed before submitting the app to the Windows Store. The first thing the process does is run the app through a version of this tool.
- *Warnings* should be addressed, although they will most likely not cause the app to be automatically rejected.
- *Passed* means you are now ready to submit the app to the Windows Store. Note that this does not guarantee that the app will be accepted, simply that it passed the basic tests.

The results you get out of WACK are actionable, if not always crystal clear. WACK does an analysis of the app binaries and runtime performance and lets you know if you have manifest problems, performance issues, crashes, forbidden API calls, and more. Visual Studio takes care of many of these for XAML apps by default.

Running WACK from the command line

Like most Visual Studio build-related tools, WACK can be run from the command line. The application name is appcert.exe and on my machine is located in C:\Program Files (x86)\Windows Kits\8.0\App Certification Kit\.

(continued)

You'll need to run the app from an administrator command prompt. If you run `appcert /?` you'll see examples of the command-line arguments. To test a Windows Store app such as ours, the command line would appear as follows (all on one line):

```
appcert test -apptype windowsstoreapp
             -packagefullname [full appx name]
             -reportoutputpath [xml file name]
```

For more information on running the WACK tool from the command line and interpreting the report data, see http://bit.ly/Win8WACK.

You can also have WACK run automatically when you create an app package for distribution to the store or for sideloading to test machines.

23.2 Sideloading for testing purposes

Although they tend to be smaller than full-blown desktop apps, Windows Store apps should still follow the same development process and lifecycle. For many of us, that includes testing by a separate team. For others, you may at least want to send the app to a friend to try out on their machine or install the app on another of your own personal machines.

Throughout my work leading up to the general availability of Windows 8 in October 2012, I received a lot of apps from partners that I tested and provided feedback on. They didn't supply them to me in source form—I was instead handed a package that could be installed using Windows PowerShell.

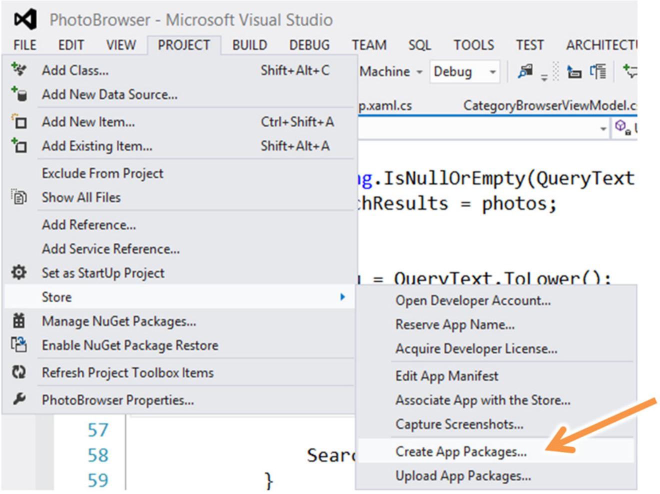
Sideloading is the process of installing Windows Store apps without going through the Windows Store. Some approaches, such as enterprise sideloading, are a primary way of deploying apps. The other more typical approach, sideloading for testing (also called developer sideloading or just sideloading), is a way to temporarily install an app for testing purposes.

Developer sideloading is only for testing with other trusted people and machines. In order to run a sideloaded package, you must have a developer account and you must accept and trust the certificate of the developer. For the average end user, this could be a security problem (the sideloaded app could be making disallowed API calls, but if you trust it, it will be allowed to do so). For testing inside an organization or group, it is acceptable.

In this section, you'll package the app for sideloading (or other deployment), learn how to get a developer license without using Visual Studio, and then install the sideloaded app from the command line.

23.2.1 Packaging an app for sideloading

When you compile and deploy in Visual Studio, no app package is created. This is different from Silverlight and other technologies, which always created the app package


PhotoBrowser - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITECTU
Add Class... Shift+Alt+C
Add New Data Source...
Add New Item... Ctrl+Shift+A
Add Existing Item... Shift+Alt+A
Exclude From Project
Show All Files
Add Reference...
Add Service Reference...
Set as StartUp Project
Store
Manage NuGet Packages...
Enable NuGet Package Restore
Refresh Project Toolbox Items
PhotoBrowser Properties...
Machine Debug
p.xaml.cs CategoryBrowserViewModel.c
Open Developer Account...
Reserve App Name...
Acquire Developer License...
Edit App Manifest
Associate App with the Store...
Capture Screenshots...
Create App Packages...
Upload App Packages...

```
ig.IsNullOrEmpty(QueryText
chResults = photos;

1 = QueryText.ToLower();
57
58              Searc
59          }
```

Create app package.

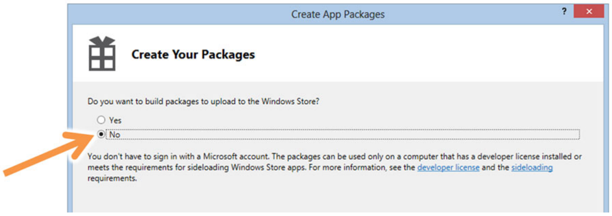
Figure 23.5 Menu option for creating an app package for sideloading or for submitting to the store

regardless. For Windows 8, the app package is a distribution mechanism only, not the final shape of the app deployment.

In order to deploy the app to the Windows Store, or to a tester for sideloading, you must explicitly package the app. From within Visual Studio, this is easily done from the Store submenu of the Project menu, as shown in figure 23.5. Make sure you have the project selected before doing this.

When you click the Create App Packages menu item, you will be presented with a wizard. By default, it will assume you're packaging for the Windows Store. To create the sideloading package, you need to select No on the first page of the wizard, as shown in figure 23.6.

Click Next on the wizard, and you'll be presented a page that lets you pick a configuration to include in the package. If the app has no C++ components (like SQLite, for example), the Neutral package is the best choice. This package will work on all CPUs. If you do include C++ components, you must build separate packages for the different architectures, because C++ is processor-specific. That's both a source of pain for developers as well as a reason why it performs so well.


Create App Packages
?
×
Create Your Packages

Do you want to build packages to upload to the Windows Store?

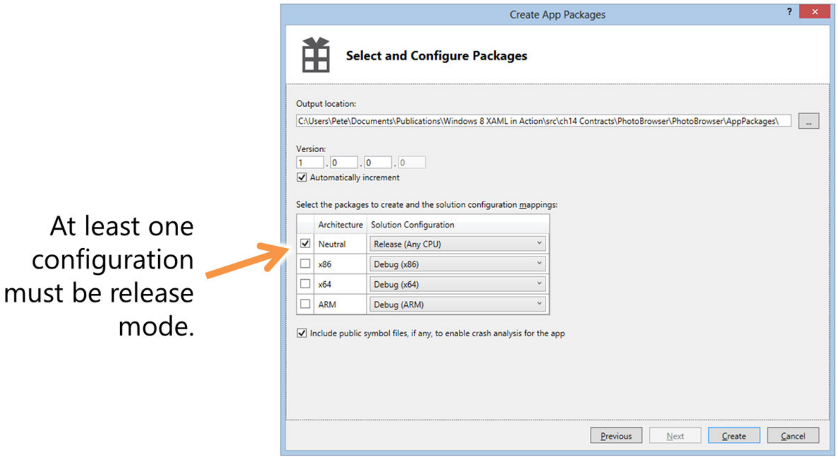
○ Yes

● No

You don't have to sign in with a Microsoft account. The packages can be used only on a computer that has a developer license installed or meets the requirements for sideloading Windows Store apps. For more information, see the developer license and the sideloading requirements.

Select No for sideloading only.

Figure 23.6 Select No if you want to create an app package for sideloading or Yes if you intend to create a package for submitting to the Windows Store.



Select and Configure Packages

Output location:
C:\Users\Pete\Documents\Publications\Windows 8 XAML in Action\src\ch14 Contracts\PhotoBrowser\PhotoBrowser\AppPackages\

Version:
1 . 0 . 0 . 0
☑ Automatically increment

Select the packages to create and the solution configuration mappings:

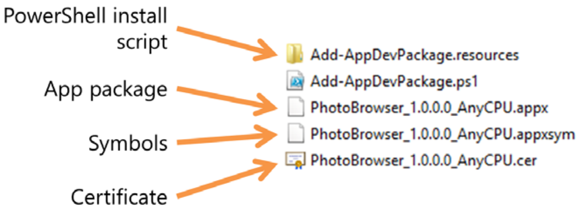
	Architecture	Solution Configuration
☑	Neutral	Release (Any CPU)
☐	x86	Debug (x86)
☐	x64	Debug (x64)
☐	ARM	Debug (ARM)

☑ Include public symbol files, if any, to enable crash analysis for the app

Figure 23.7 Be sure to pick an appropriate processor architecture and configuration. For most C# XAML apps, without C++ components, picking Neutral will be the best choice. If everything is disabled, you'll need to go back and select the project from the Solution Explorer before opening this dialog.

Figure 23.7 shows the page with the Neutral package selected. Note also that I had to change its configuration to Release or else I wouldn't be able to create the package for deployment.

Click Create and the wizard will complete, dropping a number of files into the folder specified at the top of the last wizard page. Figure 23.8 shows the files and their purposes.



Add-AppDevPackage.resources
Add-AppDevPackage.ps1
PhotoBrowser_1.0.0.0_AnyCPU.appx
PhotoBrowser_1.0.0.0_AnyCPU.appxsym
PhotoBrowser_1.0.0.0_AnyCPU.cer

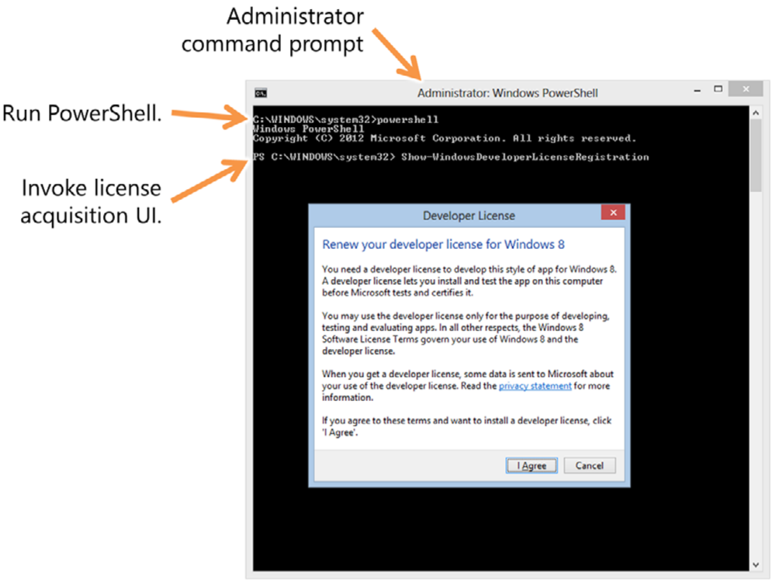
Figure 23.8 The files created by the packaging wizard. If you're missing some of these files, you may have forgotten to check the option to include public symbol files. That option isn't required but can be helpful. You'll also receive an appxupload file, which is what is used to send the app to the Windows Store.

When you distribute the app, zip up everything in this folder and send it along to the tester. They'll run the .ps1 file, but before they do that, they must get a developer license.

23.2.2 Getting a developer license without Visual Studio

In order to sideload, the user who will install the app on the test machine must have a valid and enabled developer license. They do not need a Windows Store account, just the same type of developer account that enables one to develop in Visual Studio and run locally.

When a developer first creates a Windows 8 app in Visual Studio, they are prompted to create a developer license. Of course, testers don't need to have Visual Studio installed. But without Visual Studio, how do they get the license?


Administrator command prompt
Run PowerShell.
Invoke license acquisition UI.
Administrator: Windows PowerShell
Developer License
I Agree
Cancel

```
C:\WINDOWS\system32>powershell
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS C:\WINDOWS\system32> Show-WindowsDeveloperLicenseRegistration
```

Renew your developer license for Windows 8

You need a developer license to develop this style of app for Windows 8. A developer license lets you install and test the app on this computer before Microsoft tests and certifies it.

You may use the developer license only for the purpose of developing, testing and evaluating apps. In all other respects, the Windows 8 Software License Terms govern your use of Windows 8 and the developer license.

When you get a developer license, some data is sent to Microsoft about your use of the developer license. Read the privacy statement for more information.

If you agree to these terms and want to install a developer license, click 'I Agree'.

Figure 23.9 Invoking the license acquisition UI to obtain a developer license

The first step is to have a valid Microsoft account. For most users, this will be the account they use to sign into the PC. If the user is using a local account, they can switch to a Microsoft account.

Once they have a valid Microsoft account, they can open a command prompt and run any of the following PowerShell scripts from an administrator command prompt:

- `Show-WindowsDeveloperLicenseRegistration` will invoke the Windows UI for obtaining a license. This is what you need to choose to obtain a license.
- `Get-WindowsDeveloperLicense` will tell you the expiration time and whether or not the license is valid. This command is purely informational.
- `Unregister-WindowsDeveloperLicense` will remove the developer license from the machine. Warning: don't use this unless you want your apps to stop working.

Figure 23.9 shows the command for obtaining a license being run from within PowerShell in an elevated command prompt.

Once the user has the Microsoft account and the developer license, they are able to install the sideload package.

TIP In this chapter I cover sideloading for testing purposes. There's a more involved and secure process for sideloading within the enterprise. For information on enterprise sideloading, see this TechNet page: http://bit.ly/Win8EnterpriseSideLoad.

23.2.3 Installing the sideload app package

Installing a sideload package is not the same as downloading and installing an app from the store. You have to run a Windows PowerShell script, install a certificate, and more. It's not an end-user-friendly install by any means.

Simply copy the app package files to the destination machine and double-click the .ps1 file. You'll be prompted to accept the certificate. If you're not logged in with a Microsoft account, the process is more complex, so simply keep the machine logged in using the Microsoft account when running the PowerShell script.

> **NOTE** In some cases, especially with apps that were written in C++ and then distributed along with the dependency folders, the installation will fail. Delete the dependency folders and repeat the installation.

Once the app package is installed, it will behave like any other app, but it won't have any links to the store for operations such as rating or checking for updates. What's there is sufficient for testing, which is what we're trying to do here anyway.

Before you consider putting the app in the store, it's worth looking at a technique for increasing adoption of your app: trial mode.

23.3 Enabling trial mode

Most users want to give a new app a spin before they commit to purchasing it. There's an interesting psychology where consumers who won't even bat an eye over $5 for a coffee will think long and hard about spending $1.49 for an app.

Trial mode enables you to offer a subset of functionality or full functionality for a limited time. Except in the case of some well-established brands, paid apps that also have a trial mode tend to sell far more than paid apps that offer no try-before-you-buy. Unlike other platforms, trial mode is built into the store and platform architecture; you don't need to create separate trial and full apps.

The trial period (currently perpetual, 1-, 7-, 15-, and 30-day options) is set in the Windows Store when you pick the price for your app.

To use trial mode, you must work with licensing objects from code. In this section, we'll first look at how to create mock license data for offline testing. Then, we'll look at the functions you'll use to check the license state. Finally, we'll see how you can enable a purchasing experience from directly inside the app. The listings in this section aren't meant to be added to any example app we've worked on so far but rather are meant to give you an idea of which classes to use.

23.3.1 Creating the mock license data for testing

There are two objects that come into play when working with trial mode: `CurrentApp` and `CurrentAppSimulator`. To simulate licensing scenarios for testing purposes, use the `CurrentAppSimulator` class rather than the `CurrentApp` class. `CurrentApp` goes to the Windows Store for license resolution but `CurrentAppSimulator` uses a local XML file to simulate license data. Both classes are located in the `Windows.ApplicationModel.Store` namespace.

When you use the `CurrentAppSimulator`, you'll need to create an XML file named WindowsStoreProxy.xml that contains the license data. Place this in the \Microsoft\Windows Store\ApiData subfolder tree of your app's installation folder (you'll likely need to create the entire folder structure). The format for this file is documented at http://bit.ly/Win8CurrentAppSimulator. A simple example of the license file is shown in listing 23.1.

Listing 23.1 An example license file

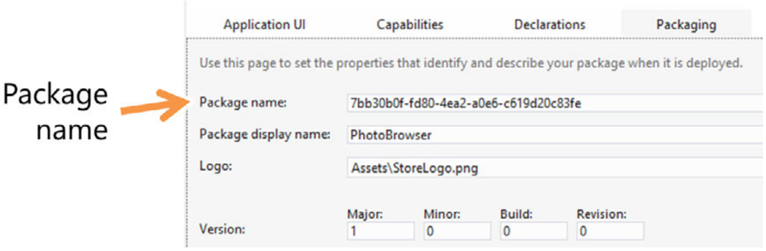
```
<?xml version="1.0" encoding="UTF-16"?>
<CurrentApp>
  <ListingInformation>
    <App>
      <AppId>7bb30b0f-fd80-4ea2-a0e6-c619d20c83fe</AppId>
      <LinkUri>http://apps.windows.microsoft.com/app/
      ➥ 7bb30b0f-fd80-4ea2-a0e6-c619d20c83fe</LinkUri>
      <CurrentMarket>en-US</CurrentMarket>
      <AgeRating>7</AgeRating>
      <MarketData xml:lang="en-us">
        <Name>Full app license</Name>
        <Description>The full license for this app.</Description>
        <Price>1.49</Price>
        <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
    </App>
  </ListingInformation>
  <LicenseInformation>
    <App>
      <IsActive>true</IsActive>
      <IsTrial>true</IsTrial>
      <ExpirationDate>2013-05-19T05:00:00.00Z</ExpirationDate>
    </App>
  </LicenseInformation>
</CurrentApp>
```

Your URI will be different, of course

The URL to the app is subject to change after the official release of Windows 8. The app's `AppId` is pulled from the Package Name field in the appx manifest designer packaging tab, as shown in figure 23.10.

The license file can be far more complex, handling licenses for individual app features, in-app purchases, and more. The MSDN page on the schema has information on how to handle the licensing for these features.

Don't leave the `CurrentAppSimulator` calls in your app when you submit it to the store or else it will fail certification. Ensure you use `CurrentApp` instead.



Use this page to set the properties that identify and describe your package when it is deployed.

Package name: 7bb30b0f-fd80-4ea2-a0e6-c619d20c83fe

Package display name: PhotoBrowser

Logo: Assets\StoreLogo.png

Version: Major: 1 Minor: 0 Build: 0 Revision: 0

Figure 23.10 The package name in the appx manifest designer

23.3.2 Checking the license state

During app startup, you'll want to check the app's license state. This is done by getting the `LicenseInformation` instance via the property by the same name on the `CurrentApp` or `CurrentAppSimulator` object. Once you have the license information object, you can check to see if the app is running in trial mode or not and, if so, when the trial mode expires. Listing 23.2 shows an example of this in action in app.xaml.cs. You don't need to put this code anywhere; it's only for illustration.

Listing 23.2 Checking the app's license state

```
private void EnableFullMode()
{
  //...
}

private void EnableTrialMode()
{
  //...
}

private LicenseInformation _licenseInformation;
private void LoadLicenseInformation()
{
  _licenseInformation = CurrentAppSimulator.LicenseInformation;

  _licenseInformation.LicenseChanged += OnLicenseChanged;
}

private void OnLicenseChanged()
{
  if (_licenseInformation.IsActive)
  {
    if (_licenseInformation.IsTrial)
    {
      EnableTrialMode();

      Debug.WriteLine("Trial mode license Expires on " +
          _licenseInformation.ExpirationDate.ToString());
    }
    else
    {
      EnableFullMode();
    }
  }
  else
  {
    // error
  }
}
```

License information

Get license information from simulator

Check for license state change

Check if active

Check if trial

Get trial expiration

This example shows how you could check at startup and also register for the `LicenseChanged` event, which will tell you when the license has expired or if the user has updated the license while the app is running.

If the license changes during the runtime of the app, you don't need to handle that. It is fine store-wise if you check the license only on startup. But if you encourage the user to purchase licenses while running your app, you'll want to check for the updated license just to provide for a better user experience.

The user may purchase the full version of your app through the store or through an app-supplied UI. If you want to increase the chance the user purchases your app, providing an in-app reminder to purchase may work, as long as it isn't a constant nag. Should you decide to go this route, you can use the `CurrentApp.RequestAppPurchaseAsync` method.

For any license to be valid, you'll need to first list the app in the Windows Store.

23.4 Listing your app in the Windows Store

So, here we are, the place this entire book has been building up to: the Windows Store. Sure, some of you may use enterprise sideloading or other mechanisms to get your apps in front of users, but if other platforms are any indicator, the vast majority of apps will be those purchased or downloaded from the Windows Store.

The specific steps required to register for a store account and list your app in the store are a bit of a moving target as the store experience is streamlined and improved. For that reason, I'll give you the high-level outline in this chapter and provide links to the appropriate MSDN reference pages for more detail.

The process for listing the app starts with getting a store account. Once you have that, you can reserve an app name, get all the listing items in order, and then list the app.

23.4.1 Getting a Windows Store account

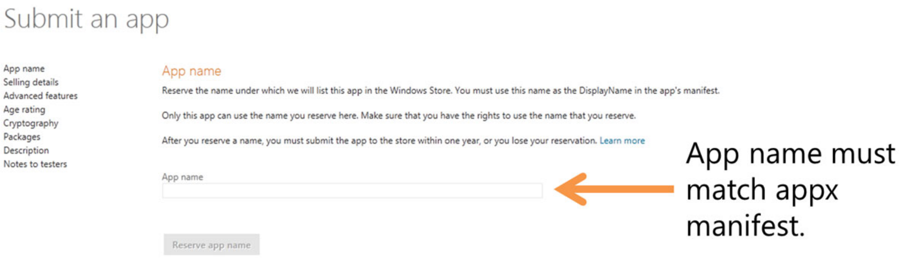
Windows Store accounts are available for individuals and businesses. Business accounts cost more but have additional capabilities, such as the ability to list desktop apps, the ability to request the Documents Library capability, and the ability to have multiple accounts associated with the store.

By now you should have a Microsoft account. It's not required to log into Windows, but it is required for getting a developer account as well as for getting a Windows Store account. I'm also going to assume you're registering for an individual Windows Store account as opposed to a business account. Before registering, get your credit card handy. Every Windows Store account, regardless of type, must be registered with a credit card.

The steps for registering for a Windows Store account, given a variety of different scenarios, are all covered on MSDN: http://bit.ly/Win8StoreAccount. The Windows Store is constantly updated, so I'll refer you to that link for the most up-to-date steps.

23.4.2 Reserving an app name

As soon as you know the name of the app you'll be working on, you should reserve that name in the store. Your app doesn't need to be finished for you to reserve the name. Reserving a name holds it for a set period (currently one year), enabling you to



Reserve the name under which we will list this app in the Windows Store. You must use this name as the DisplayName in the app's manifest.

Only this app can use the name you reserve here. Make sure that you have the rights to use the name that you reserve.

After you reserve a name, you must submit the app to the store within one year, or you lose your reservation. Learn more

App name must match appx manifest.

Figure 23.11 Reserving the name for an app. Make sure the app name matches the name in the appx manifest.

ensure no one else picks that app name. For example, I have three app names reserved in the store. Now I just need to get coding!

> **TIP** Don't reserve app names that are copyrighted or trademarked by other companies. For example, if you name your app "Facebook," expect to get your app taken down in the future when Facebook files a complaint. Additionally, names that are likely to trigger a complaint are, to the best of my knowledge, never featured or showcased in the Windows Store.

To reserve an app name, go to the Windows Store dashboard, https://appdev.microsoft.com/StorePortals/, and select the option to submit an app. You're not actually submitting anything at this point, just reserving the name. Figure 23.11 shows the page on the Windows Store dashboard.

Your name can be the same across all languages, or you can reserve names for different languages as well. The primary name must, however, match the name in the appx manifest, or else the store will reject the app submission.

23.4.3 Submitting the app for review and approval

Before you can upload your app, you'll need to provide additional selling details. This is what you'll need to figure out before you submit your app for sale:

- *Price tier*—Apps may be free or as inexpensive as $1.49. Pricing is available in $0.50 increments up to $4.99 and $1 increments up to $49.99. Beyond $49.99, the scale increases quickly to a maximum price of $999.99. Although allowed, not many $999.99 apps are going to sell in the Windows Store.
- *Trial period*—For apps that aren't free, you may pick a trial period ranging from perpetual to 1 day, 7 days, 15 days, or 30 days in duration.
- *Markets*—Be smart about where you make your app available. In some cases, an app may not be appropriate for certain markets. But pick as many markets as you can support.
- *Release date*—You can submit your app prior to a specific release date. The optional release date is the earliest date on which the app will be made available.

- *Category*—When choosing a category, use the links available on the submission page to help you pick the correct one. In addition, look at existing apps in the store and see how they've been listed.
- *Hardware requirements*—This is primarily for games. If you need a specific DirectX feature level or minimum system RAM, you can specify that here.
- *Accessibility*—If you meet accessibility guidelines, check this box so your app can be marked as being accessible.
- *Age rating*—All apps must be rated by age: 3+, 7+, 12+, or 16+. This information will be used to filter store results and to ensure apps target appropriate audiences. The ratings page has a good description for each age range including expectations for each. Games have additional ratings requirements also specified on the same page.
- *Cryptography*—If your app uses cryptography, this needs to be listed in the store. Apps that use cryptography aren't available in all markets.
- *Description*—Each app must have a description. You want this to be informative. This is what users will read when they want additional details about your app prior to downloading or purchasing it. This is the one place where most amateur apps fall down. When deciding whether to download your app, users typically look first at the screenshots and then at the description and reviews. You want a good, solid description that sells the features of your app and explains, up front, any limitations. For example, if your app is about a specific TV show, you'll want to mention up front that you can't watch the show through the app. Otherwise, you'll get a ton of one-star reviews saying they expected to be able to watch TV. Also keep in mind that the description factors into search, so you want to have the correct and appropriate terms in your description.
- *Screenshots*—Visual Studio will help you capture screenshots. Screenshots must be a minimum of 1366 x 768 (portrait or landscape) and under 2 MB each. You should show the main features of your app, with (if appropriate) meaningful data. The screenshots are almost always the first things a user looks at in the app listing, so make them count! Don't make the first screenshot the Splash screen for the app; make sure it shows your app in action. If your app is a drawing app, take a screenshot with some art from someone with artistic talent. If it's a music or video app, take the screenshot from the most dynamic part of the piece.
- *Promotional images*—There are also a number of promotional images you'll want to provide. For more information on choosing your app images and providing assets that will maximize your exposure in the store, see MSDN: http://bit.ly/Win8ChooseAppImages.
- *Privacy policy*—This is one detail that is often forgotten. Prior to the Windows 8 launch, I saw more apps rejected for a missing privacy policy than for many other reasons. If your app collects data that has the potential to be considered personal information (webcam images, contacts information, documents library access, and so on) you must provide a privacy policy explaining what you do with this information.

For the full app submission checklist, please see this MSDN page: http://bit.ly/Win8AppSubChecklist.

Once you have all of the items completed, you can upload your appx and submit the app officially for certification. To check the status of the certification process, click the Status link below the app summary. If your app has failed certification, please see this MSDN link for information on resolving certification errors: http://bit.ly/Win8StoreCertErrors.

The Windows Store submission process has a great interface that explains each step as you provide the required information for that step. When greater detail is needed, the steps link to appropriate pages on MSDN. If you've run the local Windows App Certification Kit tool, provided the necessary assets, and have an app that provides value to users, you should find the Windows Store experience pleasant and easy to understand.

Why did my app fail on resubmission?

The Windows Store is constantly improving the listing process and the certification requirements. Approval of an earlier submission isn't a guarantee of approval of a subsequent submission, even if the code and binaries are identical. You may run afoul of updated Windows Store certification policies. As an app developer, you're responsible for always adhering to the latest version of the Windows Store certification policies in effect when you submit the app.

Another cause is that a different tester may catch a problem an earlier tester missed. Additionally, testers are human, so gray areas may be interpreted differently between testers. We do our best to ensure that they all make the same decisions (something that will continue to improve as the store matures), but in the end, when asked to judge certain criteria, people will do it differently.

If the Windows Store certification team rejects the submission because of an app crash, be sure to have additional friends, family, and other guinea pigs test the app for you. An app may be smaller than a traditional desktop application, but because of the nature of user reviews and ratings, the stakes are much higher. You want a solid app from day one.

If you feel your app was unfairly or incorrectly rejected, you can open a ticket with the Windows Store team and have them help you work toward resolution. For more information on support, please see http://aka.ms/StoreSupport.

23.5 Summary

The ultimate goal of most Windows 8 app developers is to get their apps into the store and either make money from ads or app purchases or gain fame and peer respect from free downloads. Or maybe it's to get the apps out there to benefit the rest of mankind. Whatever your motivation, getting apps into the hands of users is the goal.

All apps, regardless of whether they are free, ad funded, or purchasable, require testing through the WACK tool. Before going too far down a path using a new API or technique or pulling in some native code, be sure to run the WACK tool to verify that you're playing nicely within the sandbox. It will test performance, proper API usage, and more. It won't guarantee that your app will make it into the store, but a failure will certainly guarantee it won't.

Once you've verified that the app passes the basic tests, you may want to send it to a tester to run through and verify that everything is working as it should. You can do this without involving the Windows Store by using sideloading. Sideloading is something only users who trust each other should do, because it involves accepting certificates that, in the wrong hands, could do a good job of letting malicious software onto a device.

Finally, after your testing team has given you the OK, you're ready to list the app in the Windows Store. You'll use your reserved app name, supply the privacy policy (if appropriate), pick the languages and correct stores, set prices, and then submit it for approval. The Windows Store team will run a version of the WACK tool and perform a number of tests before listing the app. Once listed, it's available for purchase.

One way to help adoption and purchases is to provide trial versions of the app. As a developer, you have complete control over what constitutes a trial version: limited time, limited functionality, or perhaps a bit of both. Regardless of which approach or approaches you take, having a trial version in the Windows Store will definitely help adoption of your app. Having balanced and appropriate trial limitations will entice the user to purchase the full version, without irritating them into throwing the app away.

With the app now in the store, listed and available for download, and the cash rolling in, you can take a bit of a breather before updating the app with enhancements or coming up with the next big idea.

And with this, we'll conclude the chapter—and this book. Together we've covered a lot, but we've only scratched the surface of what's possible in Windows. Thank you for sticking with me for this journey. Now, go build awesome apps.

index

A

accelerometer input
  AccelerometerInputService class 524–525
  code-behind for 525–526
  events for 526
  locking screen orientation 526–527
  mechanism for 523
AccelerometerInputService class 524–525
accounts, Microsoft 3
activating apps, using pinned tiles 281–284
active animations 59
ActualHeight property, UIElement class 76
ActualWidth property, UIElement class 76
Add Service Reference dialog 391, 400
AddMessage method 419, 421
AdmiralAppBar control 249
alignment, in Grid 99–101
Allow Any User to Debug option 17
AllSmallCaps 164
animations 59
app bar 31
  buttons for 432–433
  controls in
    adding buttons 246–250
    button commands 250–257
    overview 246
    styling buttons 246–250
  in PhotoBrowser example app 243–246
  in snapped view state 314–317
  menus in 261–263
  overview 241–243
  pinning 258–259
  popups for 261–263
  using GridView in 259–261
  visibility of 258–259
app manifest, declaring search intention 332–333
app name, reserving 570–571
app settings
  acting on options 553–554
  creating infrastructure for 538–542
  creating UI for 542–550
  loading 550–553
  saving 550–553
App.xaml, modifying for Share contract 330–332
AppBar button 405
AppBarButtonStyle 248
AppContainer 39
application resources 123
ApplicationData class 360–361, 537–538, 550, 552, 555
ApplicationExecutionState.Terminated 557
ApplicationPageBackgroundThemeBrush 9
ApplicationSettings class 539
ApplicationViewStates 309
AppTextBackground 126
AppTextColor 126
architecture
  for Windows 8 apps
    deployment 39–40
    driver access model 40–41
    overview 36–38
    sandbox 38
  of chat app 441–442
args class 105
args property 364
arrange pass 70–73
ArrangeOverride function 71
article element 130
Asteroids-inspired game
  overview 465–466
  Player model class 483–485

Asteroids-inspired game *(continued)*
  PlayerLocation class 483–484
  players in
    collection of 486–488
    connecting to message service 490–491
    displaying 488–489
    testing collection of 489–490
  Ship user control 517
  ships in
    adding label 475–476
    creating ship shape in Blend 472–475
    creating UserControl 471–472
  TCP message service for
    IMessageService interface 491–493
    reading location information 495–497
    sending location information 493–495
  testing 480–481, 497–498
  user controls for
    enabling rotation 477–479
    overview 476–477
    setting color 479–480
  user interface for
    orientation 469–470
    overview 466–468
    play field area 468
    view states 466–470
async keyword 12, 270, 373–374, 381
asynchronous operations
  IAsync* interfaces
    cancelling operation 380–381
    getting progress updates 378–379
    long-form asynchronous operations 376–378
    overview 373–374
    using await keyword 374–375
  importance of 371–373
  overview 369–371
  Task Parallel Library
    basic task operations 382–383
    cancelling task 384
    converting IAsync* to 386
    converting to IAsync* 385
    overview 381–382
ATAN2 function 514–515
attached properties
  for custom panels 105–107
  in XAML 61–62
AutoStartAsServer property 553
availableSize parameter 71
await keyword 12, 270, 372–374, 383
awaitable function 373–375
AwesomeButton control 54

B

Background property 130
BCL (Base Class Library) 37
binding
  change notification 186–189
  DataContext property 189–190
  modes for 185
  of UI elements 194–196
  overview 183–184
  source 184–185
  target 184–185
  using value converters 207–209
  with PasswordBox 192–193
  with TextBox 191–192
BindServiceNameAsync function 437
bitmap images 120, 137–139
Blend for Visual Studio 472–475
BooleanToVisbilityConverter class 208
BottomAppBar property 246
brushes
  ImageBrush 118–120
  LinearGradientBrush 116–118
  overview 113
  SolidColorBrush 113–116
BuildCategoryPinTile function 279
ButtonBase control 200
ButtonInTheControl 67–68
buttons
  for app bar 432–433
    adding 246–250
    commands for 250–257
    styling 246–250
  in MVVM pattern
    CheckBox 207
    HyperlinkButton 203–204
    overview 200–203
    RadioButton 204–207
bytesRetrieved property 378

C

C command 136
C++ projects, for Xbox 360 gamepad input
  compiling 534–535
  creating 527–529
  deploying 534–535
CacheMode 83
caching, subtrees for performance 83
CalculateNewLocation method 504
CalculateOrbitSpacing function 108
CameraViewModel class 178, 188, 205
CancellationToken 384
CancellationTokenSource 384
cancelling
  asynchronous operations 380–381
  tasks 384
CanGoBack property 234
CanPostNewMessage function 202, 434

canvas
  overview 87–88
  positioning in 88–89
  sizing child elements 91
  Z ordering in 89–90
Capitals property 163
Cascading Style Sheets. *See* CSS
category browser page, PhotoBrowser example app 232–235
CategoryBrowserPage function 280–281, 308, 310, 322
CategoryBrowserPage.xaml.cs file 234, 245, 324
CategoryBrowserViewModel class 232, 243, 250, 257, 279, 297, 322
CategoryPinningService class 277
certification, testing apps for 560–563
change notification, for binding 186–189
character spacing, for text 150–151
CharacterSpacing property 150
charms bar 31
chat app
  architecture of 441–442
  ChatMessage class 429, 442–444
  connecting to server
    connecting to endpoint 440
    overview 439–441
    sending data 440–441
  IMessageService interface 444–447
  listening for connections 434–439
  MainViewModel class
    connecting to server 455–456
    event handlers 457–458
    listening for connections 456
    overview 426–428
    sending chat messages 456
  TcpStreamMessageService class
    listening as server 449–450
    opening and closing connection 448–449
    overview 447–448
    processing messages 452–453
    sending messages 450–452
  UdpMessageService class
    connecting to another machine 461–462
    listening for connections 460
    overview 459–460
    parsing messages 462–463
    receiving messages 462–463
  user interface
    app bar buttons 432–433
    overview 429–430
    resources for 431
    styles for 431
    XAML for 430–431, 433–434
ChatDataService class 177
ChatMessage class 185–186, 191–192, 429, 442–444
CheckBox, in MVVM pattern 207
child elements, sizing
  in canvas 91
  in StackPanel 93
ClearType 143
CLI (Common Language Infrastructure) 41
CLR (Common Language Runtime) 37
CoCreateInstance 42
code-behind, for Twitter integration 11–13
CodePlex page 174
collection, players in Asteroids-inspired game 486–488
CollectionView 307
CollectionViewSource 219, 229
Column property 98
columns, defining for Grid 95–97
ColumnSpan property 98
COM (Component Object Model) 41, 319
  error reporting in 43
  evolution of 42–43
  overview 42
  performance improvements 43–44
Command property 201
CommandParameter property 203
CommandsRequested event 546–547
Computer Graphics category 277
ConnectAsync function 461
ConnectionReceived event 437, 450
connections, listening for 434–439
ConnectionStatus property 490
ConstructCategoryBrowserViewModel method 337
consumer apps 29–30
ContentPresenter 66
ContentTemplate property 199
ContextualAlternates property 163
ContextualLigatures property 161
ContinueWith function 378
contracts
  defined 319–320
  Search contract
    declaring in manifest 332–333
    external search requests 339–340
    in-app search requests 338–339
    overview 332
    results page for 333–338
    SearchViewModel for 333–338
  Share contract
    creating share target page 326–330
    declaring app as share target 325–326
    modifying App.xaml for 330–332
    overview 320–321
    sharing data using 321–325
control templates 130
Controller class, for Xbox 360 gamepad input 529–532

controls
  FlipView
    overview 231–232
    using 232–235
  GridView
    grouping in 226–231
    multiple rows 224–225
    single row 225–226
  in app bar
    adding buttons 246–250
    button commands 250–257
    overview 246
    styling buttons 246–250
  ListView 220–223
  PhotoBrowser example app
    category browser page 232–235
    CategoryBrowserViewModel class 232
    creating project 214–220
    ImageService class 215–217
    MainPage.xaml 235–236
    MainViewModel class 217–218
    overview 213–214
    Photo model class 215
    UI for 218–220
  semantic zoom 236–239
ControlsLib project 103
CoreWindow class 521
CreateFile method 358
CreateHttp method 411
CreateNewMessage method 192, 441
CRM (Customer Relationship Management) 276
CSS (Cascading Style Sheets) 60
CurrentApp class 567
CurrentManagedThreadId 274
custom panels
  attached property for 105–107
  creating project for 102–103
  dependency property for 103–105
  layout functions for 107–111
  OrbitPanel class 103
  overview 102
Customer Relationship Management. *See* CRM

D

d:DesignHeight property 468
d:DesignWidth property 468
Data Format field 326
data services, in MVVM pattern 176–178
data templates, for ListView 199–200
data types, specifying for REST services 411
DataContext property 189–190
DataContractJsonSerializer 412
DataTemplate statement 222
DataTemplateSelector class 351–354
DCOM (Distributed COM) 319
DDE (Dynamic Data Exchange) 319
debugging
  on remote device 14–18
  on Simulator 13–14
Declarations tab 348, 366
DecodePixelHeight 222
DecodePixelWidth 222
default tiles. *See* static tiles
default value 59
DELETE method, for REST services 415–416
DeleteAllPhotos method 257, 297
DeleteMessage method 421
DemoApp example 343–347
Dependency Injection. *See* DI
dependency properties
  for custom panels 103–105
  in XAML 58–61
DependencyObject class 58
DependencyProperty 122, 477
deploying apps
  listing in Windows Store
    getting Windows Store account 570
    reserving app name 570–571
    submitting app 571–573
  sideloading
    getting developer license 565–566
    installing sideload app package 566–567
    overview 563
    packaging app for 563–565
  testing for certification 560–563
  trial mode
    checking license state 569–570
    creating mock license data for testing 567–568
    overview 567
  Windows 8 apps 39–40
deserializing data, for REST services
  JSON 413–414
  overview 412
  XML 412–413
design inspiration for Windows Modern Style
  direct influences 21–22
  navigation in 22–23
desktops, considerations for 33
developer license, obtaining 565–566
development environment, setting up 3
device pane 5–6
Device window 6
devices
  considerations for 33
  debugging on remote 14–18
  desktops 33
  hybrid devices 34
  laptops 33
  tablets 33–34

DI (Dependency Injection) 176
DirectWrite 143
Disable method 517
DiscretionaryLigatures property 161
Dismissed event 273
Dispatcher property 74
DispatcherHelper class 437
displaying text 167
DisplayName property 215, 222, 224
Distributed COM. *See* DCOM
div element 130
DoLongRunningInitializationAsync method 273
downloading files, with HttpClient class 392–393
driver access model, for Windows 8 apps 40–41
Dynamic Data Exchange. *See* DDE
dynamic objects 413

E

elements
  binding of 194–196
  checking virtualization of 94
  in XAML 52–54
  positioning in Grid 97–99
  sizing of, and performance 84
Ellipse 137
ellipsis, for text 147–150
embedding, fonts 167–168
Enable method 517
EndPoint property 116
endpoints, connecting to 440
Enhanced Protection Mode. *See* EPM
enterprise apps 30–31
EPM (Enhanced Protection Mode) 39
error reporting, in COM 43
EvenOdd 135
EventArgs class 491
events, for accelerometer input 526
evolution, of COM 42–43
explicit styles 127–128
extended splash screens 269–275
ExtendedSplash constructor 274
ExtendedSplash.xaml page 271
ExtendedSplash.xaml.cs file 272, 282
Extensible Markup Language. *See* XML
ExterminateAppBarButtonStyle 249
external search requests 339–340
external type libraries 44

F

file pickers
  file open picker 361–363
  file picker source contract 363–368
FileInformationFactory 359
FileIO class 343, 358–359
FileOpenPicker 361
FileOpenPickerPage 363
files
  creating 355–359
  file pickers
    file open picker 361–363
    file picker source contract 363–368
  loading programmatically
    creating files and folders 355–359
    DataTemplateSelector class 351–354
    DemoApp example 343–347
    KnownFolders class 348–351
    permissions for 347–348
    StorageFiles class 348–351
    using file queries 354–355
  URI formats 359–360
FileSavePicker 361
FileTemplate property 352
filled view state 301–303
FilledShape class 132
FillRule property 134
finalSize parameter 71
FlipView 303
  overview 231–232
  using 232–235
FolderDepth property 355
FolderPicker 361
folders, creating 355–359
FolderTemplate property 352
FontFamily property 145, 163
fonts
  capitals, for OpenType fonts 163–164
  embedding 167–168
  hierarchy, for Windows apps 26
FontSize property 145, 163
FontStyle property 145
Foreground property 60, 145, 479
fractions, displaying using OpenType fonts 164–166
FrameworkElement 71, 125
FromArgb method 115
full view state 301–303
FullScreenLandscape 309

G

GamepadInputService.cs file 532
GenerateUniqueName 357
Get-WindowsDeveloperLicense 566
GetAllMessages method 421
GetAsync method 382, 384, 392
GetCategories method 281
GetChild method 64
GetChildrenCount method 64

GetDataReader method 463
GetFileFromApplicationUriAsync method 324
GetHostNames function 460
GetIsVirtualizing method 94
GetItemsAsync method 349
GetLayoutException function 73–74
GetLayoutSlot function 73
GetLocalValue function 552
GetMessages method 399
GetParent method 64
GetPhotos method 217, 288–289
GetRequestStreamAsync method 373
GetRoamingValue function 552
GetSingleMessage method 421
GetState method 530
GetThumbnailAsync method 359
GetValue method 104
GoToState method 304
governing principles, for Windows Modern Style 23–25
graphical user interface. *See* GUI
grid layout
  alignment in 99–101
  defining rows and columns 95–97
  for Windows Modern Style 27–28
  margins in 99–101
  overview 94–95
  positioning elements in 97–99
grid star sizing 97
GridView control 302
  grouping in
    at UI layer 228–231
    in model 227–228
    overview 226–227
  multiple rows 224–225
  single row 225–226
  using in app bar 259–261
GroupStyle.HeaderTemplate property 230
GroupStyle.Panel template 230
GUI (graphical user interface) 141

H

H command 136
HandleVirtualKeyDown 511
HandleVirtualKeyUp 511
Height property, UIElement class 75–76
high-DPI displays 139
HistoricalLigatures property 161
hold animations 59
HorizontalAlignment property 74, 77–78, 91, 163
HRESULTs 43
HttpClient class, downloading files with 392–393
HttpResponseMessage class 383
HttpWebRequest class 390, 411
hybrid devices, considerations for 34
HyperlinkButton, in MVVM pattern 203–204

I

IAscyncActionWithProgress interface 373
IAsync* interfaces
  cancelling operation 380–381
  converting tasks to 385
  converting to tasks 386
  getting progress updates 378–379
  long-form asynchronous operations 376–378
  overview 373–374
  using await keyword 374–375
IAsyncAction interface 373, 376
IAsyncActionWithProgress interface 378
IAsyncActionWithProgress<TProgress> interface 376
IAsyncInfo interface 373, 377
IAsyncOperation interface 373
IAsyncOperation<TResult> interface 376
IAsyncOperationWithProgress interface 373, 377–378
IAsyncOperationWithProgress<TResult,TProgress> interface 376
IDispatch interface 42
IHV (independent hardware vendor) 40
IInputService interface 502–504, 532–533
ILDASM 41, 44–45
Image property 179
ImageBrush 118–120
images, in live tiles
  creating thumbnails for 288–289
  generating notification 289–291
ImageService class 215–217, 257
IMessageService interface
  chat app 444–448
  for Asteroids-inspired game 491–493
implicit styles 130–131
in-app search requests 338–339
independent hardware vendor. *See* IHV
influences, of Windows Modern Style 21–22
infrastructure, for app settings 538–542
inheritance, for styles 128–130
InitializeLayout function 122
Inline 146–147
InlineUIContainer 156
INotifyPropertyChanged interface 186, 394
input
  accelerometer
    AccelerometerInputService class 524–525
    code-behind for 525–526
    events for 526
    locking screen orientation 526–527
    mechanism for 523

input *(continued)*
  keyboard
    code-behind for 512–513
    KeyboardInputService 508–510
    overview 507–508
    virtual keys 510–512
  making generic
    IInputService interface 502–504
    math calculations for 504–505
    overview 502
    viewmodel code for 505–507
  pointer
    code-behind for 519–521
    math calculations for 514–517
    overview 513
    PointerInputService class 517–519
  Xbox 360 gamepad
    code-behind for 534
    compiling project 534–535
    Controller class 529–532
    creating C++ project for 527–529
    deploying project 534–535
    IInputService wrapper 532–533
    overview 527–535
installing
  MVVM toolkits 174
  sideload app package 566–567
InstantMessage class 398, 400
InstantMessage.cs file 396, 398
IoC (inversion of control) 171, 176
IProgress interface 386
IsActive property 272
IsChecked property 205
IsEnabled property 179, 207, 209
IsFireButtonPressed property 504
IsMovingBackward property 504
IsOpen property 263
IsPasswordRevealButtonEnabled property 192
IsSelected property 205
IsSourceGrouped property 229
IStorageItem interface 351
ItemBackground 122
ItemClick event 307
items controls, for ListView 198–199
ItemsControl class 103, 198
ItemsPanel property 230
ItemsPanelTemplate class 489
ItemsPath property 229
ItemTemplate property 222, 224, 230
IValueConverter interface 208
IXmlNode 291

J

JSON (JavaScript Object Notation)
  deserializing 413–414
  overview 409

K

keyboard input
  code-behind for 512–513
  KeyboardInputService 508–510
  overview 507–508
  virtual keys 510–512
KeyboardInputService 508–510
keyed styles. *See* explicit styles
KnownFolders class 348–351, 359, 368

L

L command 136
language projection 36
language support, supported by WinRT 37, 47–48
laptops, considerations for 33
LaunchActivedEventArgs 556
layout
  for custom panels 107–111
  layout rounding 80–82
  margins 79–80
  multipass layout
    arrange pass 71–73
    LayoutInformation class 73–74
    measure pass 71
    overview 70–71
  padding 78–79
  performance considerations
    caching subtrees 83
    simplifying UI 82–83
    sizing of elements 84
    using PNG images instead of complex XAML 83
    using virtualization 83–84
  UIElement class
    ActualHeight property 76
    ActualWidth property 76
    Height property 75–76
    HorizontalAlignment property 77–78
    LayoutUpdated event 76
    overview 74–75
    VerticalAlignment property 77–78
    Width property 75–76
LayoutAwarePage class 303–305, 310
LayoutInformation class 71
  GetLayoutException function 73–74
  GetLayoutSlot function 73
LayoutRoot 96
LayoutUpdated event, UIElement class 76
Left property 88
license state, checking for trial mode 569–570
LicenseChanged event 569
ligatures, for OpenType fonts 160–161
LikeSelectedPhoto method 252

Line 132–134
line spacing, for text 151–153
LinearGradientBrush 57, 113, 116–118
LineHeight property 151
LineStackingStrategy property 152
ListBox control 11
ListBoxItems 64
listening for connections
  in chat app 434–439, 449–450
  using UDP sockets 460
ListView control 11, 302
  in MVVM pattern
    data templates for 199–200
    items controls for 198–199
    ObservableCollection class 197–198
    overview 197
  overview 220–223
live tiles
  images in
    creating thumbnails for 288–289
    generating notification 289–291
  overview 284–285
  queuing multiple notifications 291–294
  text in 285–288
Load method 557
LoadAsync method 438, 453, 462
Loaded event 122, 317
LoadFeedAsync function 375
loading
  app settings 550–553
  files
    creating files and folders 355–359
    DataTemplateSelector class 351–354
    DemoApp example 343–347
    KnownFolders class 348–351
    permissions for 347–348
    StorageFiles class 348–351
    using file queries 354–355
LoadMessages method 406
LoadPhotos method 228, 287, 291
local resources 121–123
local settings 541
local values 59
Location property 484
locationChanged flag 506
locking screen orientation 526–527
long-form asynchronous operations 376–378

M

M command 136
MainPage class 305
MainPage.xaml, PhotoBrowser example app 235–236
MainViewModel class
  chat app
    connecting to server 455–456
    event handlers 457–458
    listening for connections 456
    overview 426–428
    sending chat messages 456
  PhotoBrowser example app 217–218
MainViewModel.cs file 402
manifest, app 332–333
Margin property 91, 517
margins
  for elements 79–80
  in Grid 99–101
MathService class 504, 516, 519
MatrixTransform 255
Maximum property 195
measure pass 70–71
MeasureOverride method 71, 108
menus, in app bar 261–263
merging, resource dictionaries 125–127
message framing 453
MessageBox class 358
MessageDialog class 358
MessagesController 407
metadata
  in WinRT 44–46
  overview 41
Metro style
  design inspiration
    direct influences 21–22
    navigation in 22–23
  governing principles for 23–25
  grid layout for 27–28
  supporting touch in 28
  typography in 25–27
  UI elements of apps 31–33
MetroTwit 321
Microsoft accounts 3
Minimum property 195
Mode property 185
Model namespace 245, 396
Model-View-ViewModel pattern. *See* MVVM
models
  grouping in GridView 227–228
  in MVVM pattern 175–176
  sharing between client and server
    creating Modern app-compatible class library 397–398
    creating source class library 395–397
    overview 393–395
Modern app-compatible class library, creating 397–398
modes, for binding 185
Motion design 22

MoveNext method 376
ms-appx: prefix 119
multicolumn text, using RichTextBlock 157–159
multipass layout
  arrange pass 71–73
  LayoutInformation class
    GetLayoutException function 73–74
    GetLayoutSlot function 73
  measure pass 71
  overview 70–71
multiple rows, in GridView 224–225
MVVM (Model-View-ViewModel) pattern
  binding
    change notification 186–189
    DataContext property 189–190
    modes for 185
    of UI elements 194–196
    overview 183–184
    source 184–185
    target 184–185
    using value converters 207–209
    with PasswordBox 192–193
    with TextBox 191–192
  data services 176–178
  implementing in app with networking support
    creating user interface 404–406
    creating viewmodel 402–403
    overview 401–402
  model in 175–176
  MVVM toolkits
    creating project using 175
    installing 174
    overview 174
  overview 170–174
  using buttons in
    CheckBox 207
    HyperlinkButton 203–204
    overview 200–203
    RadioButton 204–207
  using ListView in
    data templates for 199–200
    items controls for 198–199
    ObservableCollection class 197–198
    overview 197
  view in 180–183
  viewmodel in 178–180
MVVM toolkits
  creating project using 175
  installing 174
  overview 174
MvvmLight template 175
MyNestedControl 67

N

namescopes 66–68
namespaces 54–55
NavigateUri property 204
navigation, in Windows Modern Style 22–23
.NET 4.5, and WinRT 48–50
NetFX30 folder 477
networking support
  MVVM implementation for
    creating user interface 404–406
    creating viewmodel 402–403
    overview 401–402
  REST services
    calling functions from client 416–421
    creating service 407–410
    DELETE method 415–416
    deserializing data 412
    deserializing JSON 413–414
    deserializing XML 412–413
    getting data from service 410–411
    overview 406–407
    POST method 415–416
    PUT method 415–416
    specifying data type 411
  sharing model between client and server
    creating Modern app-compatible class library 397–398
    creating source class library 395–397
    overview 393–395
  SOAP services
    creating service 399–400
    overview 398–399
    using service 400–401
  Windows 8 app support for
    downloading file with HttpClient class 392–393
    overview 389–390
    solution for testing 390–392
New Project dialog 175
NewMessage property 191
NonZero 135
notifications
  for live tiles
    generating 289–291
    queuing multiple 291–294
  toasts
    enabling 298
    notification service for 295–297
    overview 294–295
NotifySettingsChanged method 541
NumeralAlignment property 165
NumeralStyle property 165

O

object trees 63–66
ObservableCollection class 12, 171, 197–198, 337, 485
ObservableObject class 186
OEM (original equipment manufacturer) 40
OnActivated method 284
OnBackButtonClick event 235
OnDataChanged method 553
OneTime value 185
OneWay value 185
OnLaunched method 234, 283, 330, 552, 556–557
OnLoadDataClick event 405
OnNavigatedTo method 12, 231, 235, 245, 305, 392
OnSearchActivated 339
OnSearchQuerySubmitted handler 339
OnServicePlayerExited 490
OnServicePlayerJoined event 490, 507
OnShareTargetActivated method 330
OnSplashScreenDismissed method 283
OnSuspending method 555, 557
OnWindowCreated event 339
OPC (Open Packing Conventions) 39
Open Sound Control. *See* OSC
OpenType fonts
  displaying fractions 164–166
  font capitals 163–164
  ligatures 160–161
  stylistic sets 161–163
  subscript 166–167
  superscript 166–167
operations, asynchronous
  IAsync* interfaces
    cancelling operation 380–381
    getting progress updates 378–379
    long-form asynchronous operations 376–378
    overview 373–374
    using await keyword 374–375
  importance of 371–373
  overview 369–371
  Task Parallel Library
    basic task operations 382–383
    cancelling task 384
    converting IAsync* to 386
    converting to IAsync* 385
    overview 381–382
OrbitPanel class 102–103
OrbitPanel example
  attached property for 105–107
  creating project for 102–103
  dependency property for 103–105
  layout functions for 107–111
  OrbitPanel class 103
  overview 102
Orbits property 104
orientation
  for Asteroids-inspired game 469–470
  for StackPanel 92
  screen locking 526–527
Orientation property 92
original equipment manufacturer. *See* OEM
OSC (Open Sound Control) 40
OutputStream property 438
OverflowContentTarget property 159

P

p element 130
Package.appxmanifest file 7, 389
packaging apps, for sideloading 563–565
padding, for elements 78–79
Page element 246
page resources 121–123
pageTitle element 308
panels
  canvas
    overview 87–88
    positioning in 88–89
    sizing child elements 91
    Z ordering in 89–90
  custom panels
    attached property for 105–107
    creating project for 102–103
    dependency property for 103–105
    layout functions for 107–111
    OrbitPanel class 103
    overview 102
  Grid
    alignment in 99–101
    defining rows and columns 95–97
    margins in 99–101
    overview 94–95
    positioning elements in 97–99
  StackPanel
    orientation for 92
    overview 91–92
    sizing child elements 93
  VirtualizingStackPanel
    checking virtualization of element 94
    enabling virtualization 93–94
    overview 91–92
PasswordBox, binding for 192–193
PasswordChar property 193
passwords 552
PasswordVault class 553
Path 135–137
PathIO class 358
PCL (Portable Class Libraries) 397
pen input. *See* pointer input

pen tool 474
performance
  improvements to COM 43–44
  layout considerations
    caching subtrees 83
    simplifying UI 82–83
    sizing of elements 84
    using PNG images instead of complex XAML 83
    using virtualization 83–84
permissions, for files 347–348
Photo class
  overview 254
  PhotoBrowser example app 215
PhotoBrowser example app
  app bar in 243–246
  category browser page 232–235
  CategoryBrowserViewModel class 232
  creating project 214–220
  ImageService class 215–217
  MainPage.xaml 235–236
  MainViewModel class 217–218
  overview 213–214
  Photo model class 215
  UI for 218–220
PhotoCategory class 227, 245
Photos Browser, Windows 8 22
PickMultipleFilesAsync method 362
PickSingleFileAsync method 362
pinned tiles
  activating app using 281–284
  creating 277–281
  overview 276–277
pinning, app bar 258–259
pixels 32
play field area, for Asteroids-inspired game 468
Player model class, Asteroids-inspired game 483–485
Player property 488
PlayerConnection class 448
PlayerJoin type 446
PlayerLeave type 446
PlayerLocation class, Asteroids-inspired game 483–484
PlayerName property 478–479
players, in Asteroids-inspired game
  collection of 486–488
  connecting to message service 490–491
  displaying 488–489
  testing collection of 489–490
PNG images vs. complex XAML 83
point selection tool 473–474
pointer input
  code-behind for 519–521
  math calculations for 514–517
  overview 513
  PointerInputService class 517–519
PointerInputService class 517–519
Polyline 134–135
popups, for app bar 261–263
Portable Class Libraries. *See* PCL
positioning, in canvas 88–89
POST method, for REST services 415–416
PostNewMessageCommand 202
<prefix>:<element or attribute> syntax 55
PrepareForReading method 524–525
presentation technologies, for WinRT 47–48
PreviousExecutionState property 556
ProcessIncomingMessages function 495
progress updates, during asynchronous operations 378–379
ProgressRing control 271
projections 42, 46–47
projects
  creating 3–5
  device pane 5–6
  Solution Explorer items 7–8
  UI for, creating simple 8–9
properties, in XAML
  attached properties 61–62
  dependency properties 58–61
  property paths 62
  referencing 62
  syntax for 56–58
property paths 62
PropertyChangedEventArgs class 187
PropertyMetadata object 104
push notifications 299
PUT method, for REST services 415–416

Q

Q command 136
queries, loading files using 354–355
QuerySubmitted event 339

R

race conditions 378
RadialGradientBrush 118
RadioButton, in MVVM pattern 204–207
RaiseCanExecuteChanged method 202
RaisePropertyChanged method 187
RandomAccessStreamReference type 323
RCWs (Runtime Callable Wrappers) 46
Reactive Extensions. *See* RX
ReadAsStringAsync method 383
ReadingChanged event 526
RecordSet class 416

Rectangle 137
Register method 106
RegisterAttached method 106
RelayCommand 201
remote debugger 15
Remote Debugging Monitor 17
remote devices, debugging on 14–18
Remote Procedure Call. *See* RPC
Rendering event 72
RenderingEventArgs 72
RenderTransform 255, 478
ReportDataRetrieved method 331
ReportError method 331
ReportInterval event 526
ReportStarted method 331
ReportSubmittedBackgroundTask method 331
Representational State Transfer service. *See* REST
RequestAppPurchaseAsync method 570
reserving app name 570–571
resource dictionaries
  merging 125–127
  overview 123–125
ResourceDictionary block 123, 209
resources
  application resources 123
  local resources 121–123
  overview 120–121
  page resources 121–123
  resource dictionaries
    merging 125–127
    overview 123–125
Resources block 209
REST (Representational State Transfer) services
  calling functions from client 416–421
  creating service 407–410
  DELETE method 415–416
  deserializing data
    JSON 413–414
    overview 412
    XML 412–413
  getting data from service 410–411
  overview 406–407
  POST method 415–416
  PUT method 415–416
  specifying data type 411
results page, for Search contract 333–338
resuming app 555–557
RetrievalProgress 378
Rich Text Format. *See* RTF
RichEditBox 154
RichTextBlock 154–156
  multicolumn text using 157–159
  overview 153–157
RichTextBlockOverflow 158–159
RoActivateInstance 42
roaming settings 541
RotateTransform 255, 478
rotation, enabling for Asteroids-inspired game 477–479
rover remote control example. *See* Model-View-ViewModel pattern
Row property 98
RowDefinitions property 262
rows
  defining for Grid 95–97
  in GridView
    multiple 224–225
    single 225–226
RowSpan property 98
RPC (Remote Procedure Call) 406
RTF (Rich Text Format) 154
Run method 273
runtime broker 37
Runtime Callable Wrappers. *See* RCWs
RX (Reactive Extensions) 372

S

S command 136
sandbox, for Windows 8 apps 38
ScaleTransform 255
screen orientation, locking 526–527
Search contract
  declaring in manifest 332–333
  external search requests 339–340
  in-app search requests 338–339
  overview 332
  results page for 333–338
  SearchViewModel for 333–338
SearchResultsPage 333, 337
SearchViewModel, for Search contract 333–338
secondary tiles. *See* pinned tiles
SecondaryTile class 277
Segoe UI Light 268
SelectedCamera property 205
SelectedItem property 312, 418
SelectedMessage property 418
SelectionMode attribute 418
SelectTemplateCore method 352
semantic zoom 236–239
SemanticZoom control 260, 304
SendChatMessage function 460
SendLocationUpdate method 493
SendPlayerJoinMessage function 460
SendPlayerLeaveMessage function 460
SendRequest method 418–419
servers, connecting to
  connecting to endpoint 440
  overview 439–441
  sending data 440–441

services, for toast notifications 295–297
SetAttribute 291
SetDataProvider method 323
settings, app
  acting on options 553–554
  creating infrastructure for 538–542
  creating UI for 542–550
  loading 550–553
  saving 550–553
SettingsChangedRemotely event 539
SettingsConnectivityViewModel 540
SettingsOptionsPage 543
SettingsOptionsViewModel class 539–541
SettingsPane class 549
SettingsPaneService class 546
SettingsService class 539–541, 554
SetUpInputHandling method 437–438, 441
SetValue method 104
Shaken event 526
Share contract
  creating share target page 326–330
  declaring app as share target 325–326
  modifying App.xaml for 330–332
  overview 320–321
  sharing data using 321–325
ShareOperation property 330
ShareTargetPage 327, 330
ShareTargetViewModel class 328
Ship.xaml.cs file 471
ships, Asteroids-inspired game
  adding label 475–476
  creating ship shape in Blend 472–475
  creating UserControl 471–472
Show Grid control 8
Show method 546
Show-WindowsDeveloperLicenseRegistration 566
sideloading
  getting developer license 565–566
  installing sideload app package 566–567
  overview 563
  packaging app for 563–565
Simple Object Access Protocol services. *See* SOAP
Simulator, debugging on 13–14
single row, in GridView 225–226
sizing, child elements
  in canvas 91
  in StackPanel 93
SkewTransform 255
SlashedZero property 165
SmallCaps 164
Snap to Grid control 8
snapped view state
  app bar in 314–317
  creating 305–307
  detail pages in 309–314
  overview 301–303
SnappedPageHeaderTextStyle 308
SnapsTo property 195
SOAP (Simple Object Access Protocol) services
  creating service 399–400
  overview 398–399
  using service 400–401
sockets
  architecture of 441–442
  ChatMessage class 429, 442–444
  connecting to server
    connecting to endpoint 440
    overview 439–441
    sending data 440–441
  IMessageService interface 444–447
  listening for connections 434–439
  MainViewModel class
    connecting to server 455–456
    event handlers 457–458
    listening for connections 456
    overview 426–428
    sending chat messages 456
  overview 423–425
  TcpStreamMessageService class
    listening as server 449–450
    opening and closing connection 448–449
    overview 447–448
    processing messages 452–453
    sending messages 450–452
  UDP sockets 458–459
  UdpMessageService class
    connecting to another machine 461–462
    listening for connections 460
    overview 459–460
    parsing messages 462–463
    receiving messages 462–463
  user interface
    app bar buttons 432–433
    overview 429–430
    resources for 431
    styles for 431
    XAML for 430–431, 433–434
solid state drives. *See* SSD
SolidColorBrush 113–116, 484
Solution Explorer, items in 7–8
SortElements function 108–109
source class library, creating 395–397
spacing
  character spacing 150–151
  line spacing 151–153
spell checking, in TextBox control 193–194
splash screens
  extended 269–275
  overview 267
  static 267–269
SplashScreen class 270, 272–273

SplashScreen.png 268
SQLite 368
src attribute 290
SSD (solid state drives) 522
StackOverflow 204
StackPanel 53
  orientation for 92
  overview 91–92
  sizing child elements 93
StandardGradient 122
StandardLigatures property 161
StandardStyles.xaml file 7, 125, 246, 364
star sizing 97
StartLayoutUpdates handler 317
StartNew method 382, 438
StartPoint property 116
states, for Windows 8 apps 31–33
static splash screens 267–269
static tiles 275–276
StaticResource statement 222
StepFrequency 195
StopLayoutUpdates handler 317
StorageFile class 345, 348, 359, 361, 368
StorageFolder class 343, 349, 354–355, 359, 368
StorageItemDataTemplateSelector class 352
StoreAsync method 439, 441
Stretch property 119
StrokeDashArray property 133–134
style setters 59
Style tag 60
styles
  explicit styles 127–128
  implicit styles 130–131
  inheritance for 128–130
  overview 127
stylistic sets, for OpenType fonts 161–163
StylisticSetN property 163
subscript, for OpenType fonts 166–167
superscript, for OpenType fonts 166–167
support
  MVVM implementation for
    creating user interface 404–406
    creating viewmodel 402–403
    overview 401–402
  REST services
    calling functions from client 416–421
    creating service 407–410
    DELETE method 415–416
    deserializing data 412
    deserializing JSON 413–414
    deserializing XML 412–413
    getting data from service 410–411
    overview 406–407
    POST method 415–416
    PUT method 415–416
    specifying data type 411
  sharing model between client and server
    creating Modern app-compatible class library 397–398
    creating source class library 395–397
    overview 393–395
  SOAP services
    creating service 399–400
    overview 398–399
    using service 400–401
  Windows 8 app support for
    downloading file with HttpClient class 392–393
    overview 389–390
    solution for testing 390–392
suspending app 554–555
Swiss Design 21
SynchronizationContext object 273
System.Diagnostics namespace 393
System.Linq.* namespace 49
System.Net.* namespace 49
System.Net.Http namespace 373, 393
System.Runtime.Serialization.Json namespace 49
System.Runtime.Serialization.Xml namespace 49
System.Runtime.WindowsRuntime namespace 49, 385
System.Runtime.WindowsRuntime.UI.Xaml namespace 50
System.Threading.Tasks.* namespace 50, 270

T

T command 136
tablets, considerations for 33–34
target
  declaring app as for sharing 325–326
  for binding 184–185
TargetProperty 62
Task method 270
Task Parallel Library. *See* TPL
TaskFactory.StartNew method 273
TCP message service, for Asteroids-inspired game
  IMessageService interface 491–493
  listening as server 449–450
  opening and closing connection 448–449
  overview 447–448
  processing messages 452–453
  reading location information 495–497
  sending location information 493–495
  sending messages 450–452
templated properties 59
TestAddingPlayers method 490
testing
  apps for certification 560–563
  Asteroids-inspired game 497–498
  on remote device 14–18
  on Simulator 13–14
  overview 13

TestPositionUpdate function 497
text
  character spacing 150–151
  displaying 167
  ellipsis for 147–150
  embedding fonts 167–168
  in live tiles 285–288
  Inline 146–147
  line spacing 151–153
  OpenType fonts
    displaying fractions 164–166
    font capitals 163–164
    ligatures 160–161
    stylistic sets 161–163
    subscript 166–167
    superscript 166–167
  overview 141–143
  RichTextBlock
    multicolumn text using 157–159
    overview 153–157
  spacing 167
  TextAlignment property 147–150
  TextBlock 144–146
  TextWrapping property 147–150
TextAlignment property 147–150
TextBlock element 53, 144–146
TextBox control
  autocorrect in 193–194
  binding for 191–192
  spell checking in 193–194
TextTrimming property 142, 148
TextWrapping property 147–150
ThemeResources.xaml file 124–125
Thread.Sleep function 506
thumbnails, for live tiles 288–289
ThumbnailUri property 288–289
TileNotificationService class 285, 290–292
tiles, app
  live
    images in 288–291
    overview 284–285
    queuing multiple notifications 291–294
    text in 285–288
  pinned
    activating app using 281–284
    creating 277–281
    overview 276–277
  static 275–276
TileSquareBlock template 285
TileWideImageCollection 289
toast notifications
  enabling 298
  notification service for 295–297
  overview 294–295
ToastNotificationService class 295
ToggleButton 201
Top property 88
TopAppBar property 246
totalBytesToRetrieve property 378
touch input. *See* pointer input
touch support, in Windows Modern Style 28
TPL (Task Parallel Library) 372
  basic task operations 382–383
  cancelling task 384
  converting IAsync* to 386
  converting to IAsync* 385
  overview 381–382
TransformGroup 255
TranslateTransform 255
traversing, visual trees 64–66
trial mode
  checking license state 569–570
  creating mock license data for testing 567–568
  overview 567
TTF (True Type Font) 168
Tweet class 10
Twitter, integrating in app
  code-behind for 11–13
  displaying in app 10–11
  overview 9–10
  Tweet class 10
TwoWay value 185
TypeName.AttachedPropertyName syntax 61
typography, in Windows Modern Style 25–27

U

UDP (User Datagram Protocol) 458–459
UdpMessageService class, chat app
  connecting to another machine 461–462
  listening for connections 460
  overview 459–460
  parsing messages 462–463
  receiving messages 462–463
UI (user interface)
  binding of elements in 194–196
  creating simple 8–9
  elements in Windows 8 apps 31–33
  for app settings 542–550
  for Asteroids-inspired game
    orientation 469–470
    overview 466–468
    play field area 468
    view states 466–470
  for chat app
    app bar buttons 432–433
    overview 429–430
    resources for 431
    styles for 431
    XAML for 430–431, 433–434

  grouping in GridView 228–231
  layout rounding 80–82
  margins 79–80
  multipass layout
    arrange pass 71–73
    LayoutInformation class 73–74
    measure pass 71
    overview 70–71
  padding 78–79
  performance considerations
    caching subtrees 83
    simplifying UI 82–83
    sizing of elements 84
    using PNG images instead of complex XAML 83
    using virtualization 83–84
  PhotoBrowser example app 218–220
  simplifying for performance 82–83
  UIElement class
    ActualHeight property 76
    ActualWidth property 76
    Height property 75–76
    HorizontalAlignment property 77–78
    LayoutUpdated event 76
    overview 74–75
    VerticalAlignment property 77–78
    Width property 75–76
UIElement class
  ActualHeight property 76
  ActualWidth property 76
  Height property 75–76
  HorizontalAlignment property 77–78
  LayoutUpdated event 76
  overview 74–75
  VerticalAlignment property 77–78
  Width property 75–76
UnconsumedBufferLength property 453
uniform resource identifier. *See* URI
Unloaded event 317
Unregister-WindowsDeveloperLicense 566
Update method 286
UpdateChildren method 345, 355
UpdateChildrenWithQuery 355
UpdateMessage method 421
UpdateSearchResults method 337
UpdateSelectedMessage 419
UpdateState method 519
URI (uniform resource identifier) 359–360
uriRoot variable 324
UseLayoutRounding property 81
user controls, for Asteroids-inspired game
  enabling rotation 477–479
  overview 476–477
  setting color 479–480
User Datagram Protocol. *See* UDP
user interface. *See* UI
UserInformation class 295, 455
UserProfile namespace 297

V

V command 136
value converters, using for binding 207–209
value precedence 58–59
value syntax 57
Values dictionary 552
VariableSizedWrapGrid 93
VariableSizedWrappedGrid 225
Variants property 166
vector graphics
  Ellipse 137
  Line 132–134
  overview 132
  Path 135–137
  Polyline 134–135
  Rectangle 137
VerticalAlignment property, UIElement class 77–78
view controls
  FlipView
    overview 231–232
    using 232–235
  GridView
    grouping in 226–231
    multiple rows 224–225
    single row 225–226
  ListView 220–223
  PhotoBrowser example app
    category browser page 232–235
    CategoryBrowserViewModel class 232
    creating project 214–220
    ImageService class 215–217
    MainPage.xaml 235–236
    MainViewModel class 217–218
    overview 213–214
    Photo model class 215
    UI for 218–220
  semantic zoom 236–239
view states
  filled 301–303
  for Asteroids-inspired game 466–470
  full 301–303
  LayoutAwarePage class 303–305
  overview 300–301
  snapped
    app bar in 314–317
    creating 305–307
    detail pages in 309–314
    overview 301–303
  using visual states for 307–309

view, in MVVM pattern 180–183
ViewBox element 75
ViewModel folder 328, 335
viewmodel, in MVVM pattern 178–180, 217–218
ViewModelBase class 214
virtual keys 510–512
virtualization
  checking for element 94
  enabling 93–94
  using for performance 83–84
VirtualizingStackPanel 93–94
  checking virtualization of element 94
  enabling virtualization 93–94
  overview 91–92
Visibility property 172, 309
visibility, of app bar 258–259
Visual State Manager. *See* VSM
visual states, using for view states 307–309
visual trees, traversing 64–66
VisualStateGroup 308
VisualStateManager tag 307
VisualTreeHelper class 64
VSM (Visual State Manager) 307

W

WACK (Windows App Certification Kit) tool 560–563
web host 37
WebSockets 463
Width property, UIElement class 75–76
Window.Current 273
Windows 8 apps
  architecture for
    deployment 39–40
    driver access model 40–41
    overview 36–38
    sandbox 38
  consumer apps 29–30
  device considerations for
    desktops 33
    hybrid devices 34
    laptops 33
    overview 33
    tablets 33–34
  enterprise apps 30–31
  networking support in
    downloading file with HttpClient class 392–393
    overview 389–390
    solution for testing 390–392
  overview 28–29
  states for 31–33
  UI elements in 31–33
Windows App Certification Kit tool. *See* WACK
Windows core 37
Windows Modern Style
  design inspiration
    direct influences 21–22
    navigation in 22–23
  governing principles for 23–25
  grid layout for 27–28
  supporting touch in 28
  typography in 25–27
  UI elements of apps 31–33
Windows Presentation Foundation. *See* WPF
Windows Runtime. *See* WinRT
Windows Store
  getting Windows Store account 570
  reserving app name 570–571
  submitting app 571–573
Windows.ApplicationModel.Store namespace 567
Windows.Data.Json namespace 49
Windows.Data.Xml.* namespace 49
Windows.Devices.* namespace 49
Windows.Foundation namespace 49, 373
Windows.Media.* namespace 49
Windows.Networking.* namespace 49
Windows.Security.* namespace 49
Windows.Storage namespace 348, 359
Windows.Storage.* namespace 49
Windows.Storage.Pickers namespace 362
Windows.Storage.Streams namespace 323
Windows.System.UserProfile namespace 455
Windows.UI.Colors class 115
Windows.UI.StartScreen 277, 280
Windows.UI.Xaml.* namespace 49
Windows.UI.Xaml.Markup.XamlReader class 115
Windows.Web.Syndication namespace 49
Windows.winmd file 45
WindowsRuntimeSystemExtensions class 385
WinRT (Windows Runtime)
  and .NET 4.5 48–50
  COM
    error reporting in 43
    evolution of 42–43
    overview 42
    performance improvements 43–44
  languages supported 47–48
  metadata for 44–46
  overview 41–42, 48–50
  presentation technologies for 47–48
  projections 46–47
WireMessageType enum 492
WPF (Windows Presentation Foundation) 51
WriteLine method 274
WritePascalStyleString method 495
WritePlayerLocation method 495

X

x:Class property 54
x:Name property 54
XAML (Extensible Application Markup Language)
  elements 52–54
  namescopes 66–68
  namespaces 54–55
  object trees 63–66
  overview 51–52
  properties
    attached properties 61–62
    dependency properties 58–61
    property paths 62
    referencing 62
    syntax for 56–58
XamlReader.Load API 68
Xbox 360 gamepad input
  code-behind for 534
  compiling project 534–535
  Controller class 529–532
  creating C++ project for 527–529
  deploying project 534–535
  IInputService wrapper 532–533
  overview 527–535
XINPUT_STATE structure 529
XML (Extensible Markup Language), deserializing 412–413
xml:space attribute 156
XmlElement 291
xmlns:x statement 54
XmlSerializer class 412–413
XPosition property 179

Y

YPosition property 179

Z

Z ordering, in canvas 89–90
ZIndex property 88, 90, 92
zoom, semantic 236–239

WINDOWS/.NET

Windows Store App Development

Pete Brown

The Windows Store provides an amazing array of productivity tools, games, and other apps directly to the millions of customers already using Windows 8.x or Surface. Windows Store apps boast new features like touch and pen input, standardized app-to-app communication, and tight integration with the web. And, you can build Windows Store apps using the tools you already know: C# and XAML.

Windows Store App Development introduces the Windows 8.x app model to readers familiar with traditional desktop development. You'll explore dozens of carefully crafted examples as you master Windows features, the Windows Runtime, and the best practices of app design. Along the way, you'll pick up tips for deploying apps, including selling through the Windows Store.

What's Inside

- Designing, creating, and selling Windows Store apps
- Developing touch and sensor-centric apps
- Working C# examples, from feature-level techniques to complete app design
- Making apps that talk to each other
- Mixing in C++ for even more features

This book requires some knowledge of C#. No experience with Windows 8 is needed.

Pete Brown is a Developer Evangelist at Microsoft and author of *Silverlight 4 in Action* and *Silverlight 5 in Action*.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/WindowsStoreAppDevelopment

Free eBook
SEE INSERT

“Informative, fun, and easy to read.”

—Todd Miranda
NxtDimension Solutions

“Broad coverage of all aspects of W8 XAML development.”

—Roland Civet, iSolutions For You!

“Pete is a consistently great author, and once again he nails his subject.”

—Gordon Mackie Openfeatured Ltd.

“Your roadmap to modern Windows design.”

—Patrick Toohey
Mettler-Toledo Hi-Speed

“Much less a book than a must-have tool for efficient and quality app development.”

—Dave Campbell, WynApse

ISBN 13: 978-1-617290-94-7
ISBN 10: 1-617290-94-7
54999
9 781617 290947

MANNING

$49.99 / Can $52.99 [INCLUDING eBOOK]